



E.T.S. de Ingenieros en Informática
Universidad Politécnica de Madrid

Algoritmos y Estructura de Datos

Autores: Pablo Nogueira Iglesias
Guillermo Román Díez

Grado en Ingeniería Informática
Grado en Matemáticas e Informática
Doble Grado en Ingeniería Informática y ADE

10 de Septiembre de 2018

Índice general

1	Abstracción de Datos	7
1.1	Características del lenguaje Java para la abstracción de datos	7
1.1.1	Conceptos de programación imperativa que hay que tener claros	7
1.2	Introducción a los TADs y los algoritmos	25
2	Listas Indexadas	31
2.1	Interfaz <code>IndexedList</code>	31
3	Pilas (LIFO) y Colas (FIFO)	33
3.1	Pila - LIFO (Last In First Out)	33
3.2	Colas FIFO (First In First Out)	35
4	Complejidad	37
5	Ordenación	41
5.1	Órdenes Totales	41
5.2	Interfaces <code>Comparable</code> y <code>Comparator</code>	42
5.2.1	Interfaz <code>java.lang.Comparable</code>	42
5.2.2	Interfaz <code>java.util.Comparator</code>	43
6	Listas de Posiciones	47
6.1	Los interfaces: <code>Position</code> y <code>PositionList</code>	48
6.2	Implementación mediante cadenas de nodos doblemente enlazados	56
7	Iteradores	61
8	Recursividad	75

Importante

- Lo primero de todo decir que ESTO NO SON UNOS APUNTES COMPLETOS. Se trata de un *guion* elaborado por Pablo Nogueira y Guillermo Román, con los temas a tratar en las clases de teoría de la asignatura *Algoritmos y Estructuras de Datos*. Incluye referencias bibliográficas, definiciones y explicaciones útiles, ejemplos de código para analizar y discutir en clase, etc.
- El guión no contiene todo el material docente necesario para superar la asignatura. Son fundamentales las explicaciones y el desarrollo de código en clase, las transparencias, y las librerías de código descargables desde el Moodle.

1. Abstracción de Datos

1.1 Características del lenguaje Java para la abstracción de datos

1.1.1 Conceptos de programación imperativa que hay que tener claros

Comandos

Los comandos o mandatos afectan el estado del programa, es decir, el flujo de control o los valores de las variables. Son comandos:

- Las expresiones seguidas de “;”. Por ejemplo: la asignación, la invocación de un método, la creación de un objeto con **new**, expresiones de incremento y decremento, etc. Algunas expresiones cambian el estado del programa.
- Declaraciones e inicializaciones de variables — no confundir estas últimas con asignaciones.
La inicialización por defecto es sólo para atributos de clase (la hace el constructor por defecto de la clase). No hay inicialización por defecto para variables locales a un método o a un bloque. Éstas tienen que tener valor obligatoriamente antes de usarse. Para ello pueden inicializarse o asignarse antes de usarse en una expresión.
- Bloques: comienzan por “{” y terminan con “}”. Se permiten bloques vacíos (aunque está feo en muchos casos) y anidados (*nesting*).
- Flujo de control:
 - Condicional: if-then, if-then[-else], switch. (¿Qué es if-else-if?)
 - Bucles: while, do-while, for.

Equivalencia “while” y “do-while”:

<pre>INIT while (COND) { BODY }</pre>	<pre>INIT do { BODY } while (COND);</pre>
-------------------------------------------	-----------------------------------------------

El “for” es un tipo de “while”:

<pre>for (INIT ; COND ; POST) { BODY }</pre>	<pre>INIT while (COND) { BODY POST }</pre>
--------------------------------------------------	----------------------------------------------------

- Otros: return, throw, try-catch-finally, assert.
Todos ellos pueden anidarse, pero después de algunos no tiene sentido poner sentencias (el compilador se queja si las pones).
- Mandato vacío (“;”) (se puede usar en bucles)

Las variables tienen un tipo (*type*) y un ámbito (*scope*). El tipo se indica al declarar la variable. El ámbito es el trozo del programa en el que la variable es visible. Concepto importante: “shadowing” (ensombrecido). El lenguaje Java tiene unas reglas de ámbito y visibilidad establecidas. Algunos compiladores o entornos integrados como Eclipse pueden ser más restrictivos.

Ejercicio 1.1 Determina el ámbito de las distintas declaraciones de la variable 'x':

```
public class C {
    int x;

    public C(int x) { x = x; }

    public void m1(int x) {
        for (int x = 1; x < 10; x++) {
            x++;
        }
        if (x < 0) {
            int x = 1;
            this.x = x;
        }
    }

    public void m2(int c) {
        x = c;
    }
}
```

Entornos como “Eclipse” pueden protestar sin razón por algo como esto:

```
int x; // no está inicializada

if (2 == 1 + 1) {
    x = 1;
}

System.out.println(x); // Eclipse se queja aquí
```

Expresiones

Las expresiones se evalúan a un valor de un tipo y se pueden asignar en variables que sean del mismo tipo. Son expresiones los literales, las constantes (entre ellas **this**, **super**, **null**), las variables (atributos de clase, parámetros de métodos, variables locales a un bloque), operadores aplicados a argumentos, la invocación de un método, el upcasting, etc. Por ejemplo, el siguiente fragmento es una expresión booleana:

```
(a = b = 1 + c * y.scale() - ((Automovil)Vehiculo).getPrice()) != 0
```

Hay dos familias de tipos: **primitivos** (o básicos) y **referencias** (arrays, clases, interfaces, enumerados).

- Las variables de *tipo básico* almacenan directamente el valor al que son inicializadas o asignadas.
- Las variables de *tipo referencia* almacenan una referencia o puntero al (es decir, la dirección de memoria del) objeto o array, cuyos datos están situados en otra parte de la memoria.

Operadores

Los **operadores** pueden ser aritméticos, lógicos (estrictos o en cortocircuito), el operador condicional, el operador de asignación, el operador “.” de acceso a un campo de un objeto (atributo o método), el operador de acceso a un elemento de un “array”, el operador **new**, **instanceof**, etc.

Recordad que los **operadores lógicos** evaluados en **cortocircuito** son diferentes a los operadores de la lógica convencional. **A && B** no es la conjunción de los predicados ‘A’ y ‘B’. Su significado es “si A es cierto entonces el valor de **A && B** es el valor de B, pero si A es falso entonces el valor de **A && B** es falso”. Más resumido:

A && B = “si A entonces B pero si no A entonces no A (y B o no B)”

A || B = “si A entonces A (y B o no B) pero si no A entonces B”

Entre otras cosas, esto permite en **&&** usar A como condición preliminar a la comprobación de B, y en **||** usar !A como condición preliminar a la comprobación de B.

Por ejemplo, si ‘v’ es null entonces ‘v[i]’ lanza ‘NullPointerException’ pero estas dos expresiones no lanzan la excepción. La primera es falsa si ‘v’ es null mientras que la segunda es cierta.

```
v != null && v[i] == 0
```



```
v == null || v[i] == 0
```

Veremos más abajo las leyes de *de Morgan* en cortocircuito.

Los operadores tienen una **precedencia** y una **asociación** (no confundir con *asociatividad* que es la propiedad de que la asociación del operador no afecta el resultado). Conviene conocer la tabla de precedencia y asociación de Java para en caso de que no haya paréntesis explícitos que indiquen dicha precedencia o asociación. La expresión anterior es equivalente a la siguiente según las tablas de precedencia y asociación de Java y los paréntesis son por tanto innecesarios, pero si ayudan a la comprensión del código es mejor ponerlos:

```
(a = (b = ((1 + (c * (y.scale())))-(Automovil)Vehiculo).getPrice())) != 0
```

Ejercicio 1.2 Dibuja las variables como cajas y dibuja dentro los valores que toman durante la ejecución.

```
public static void main(String [] args) {
    int v1 = 3;
    int v2 = 5;
    Object o = new Object ();
    Object o2 = new Object ();

    v3 = v1;
    v1 = v2 = 6;
    v1 = 6 = v2;    // Error

    o = o2; // que pasa con o2? y el objeto al que apuntaba?
}
```

A continuación vemos un ejemplo de código para entender la asignación de variables de tipo `int`.

```
public static void main(String [] args) {
    int v1 = 3;
    int v2 = v1;

    if (v1 == v2) { // cierto
        System.out.println("Las variables almacenan el mismo valor");
    }
    v2 = 3;

    if (v1 == v2) { // cierto
        System.out.println("Las variables almacenan el mismo valor");
    }
    v2 = 4;

    if (v1 == v2) { // falso
        System.out.println("Las variables almacenan el mismo valor");
    }
}
```

Arrays

Los arrays, que también llamaremos “vectores” en el guión. (No confundirlos con los `Vector<E>` de la Java Collection Framework.)

Ejercicio 1.3 Dibuja con cajas y flechas las variables y los arrays que resultan de estas declaraciones.

```
int [] a;

int [] a2 = null;

int [] b = new int[20];

b[0] = 7;
```

```

int [] c = new int[getIntegerFromUser()];

String [] d = new String[10];    // no hay ningún String todavía

d[0] = new String(''Hola'');

int [] e = new int[0];

int [] f = { };

int [] g = { 7, 9, 8 };

Clase [] h = new SubClase[5];

```

Es fundamental conocer la condición de parada o de continuación de un bucle y utilizarla para hacer código de mejor calidad. Aplicar leyes de *de Morgan*, aprovechar el cortocircuito, etc. Por ejemplo, el método indica si el array de enteros 'arr' tiene un 0.

```

public static boolean hayCero(int arr []) {
    int i;
    for (i = 0; i < arr.length && arr[i] != 0; i++)
        ; // El bucle no ejecuta nada
    return i < arr.length;
}

```

No es necesario que el bucle tenga ninguna sentencia. El bucle termina cuando el índice 'i' está fuera de rango (es mayor o igual que el tamaño del array) o bien cuando está en rango pero se encuentra el primer cero. Ambas condiciones son excluyentes.

```
i >= arr.length || i < arr.length && arr[i] == 0
```

La condición de terminación es la negada de la de continuación del bucle, la una se obtiene de la otra usando las **leyes de de Morgan en cortocircuito**:

```

! (A && B)           = !A || A && !B
! (!A || A && ! B)  = A && B

```

La condición `i < arr.length` es verdadera en el segundo caso donde también es verdadero que se ha encontrado el cero. Por tanto basta con devolver el valor de verdad de esa expresión.

Se prefiere el bucle anterior al siguiente que usa una variable booleana innecesaria. Se puede comenzar usando la variable booleana pero es preferible sustituir variables booleanas por las expresiones que significan. El uso de variables booleanas puede ser peligroso (al confundirlas) y hacer menos legible el programa. Este código sería una opción razonable:

```

boolean encontrado = false;
for (int i = 0; i < arr.length && !encontrado; i++) {
    encontrado = arr[i] == 0;
}
return encontrado;

```

Lo siguiente es equivalente pero algo peor:

```

boolean encontrado = false;
for (int i = 0; i < arr.length && !encontrado; i++) {
    if (arr[i] == 0) {
        encontrado = true;
    }
}
return encontrado;

```

Lo siguiente todavía peor:

```

boolean encontrado = false;
for (int i = 0; i < arr.length && !encontrado; i++) {
    if (arr[i] == 0) {

```

```

        encontrado = true;
    }
    else {
        encontrado = false
    }
}
return encontrado;

```

Y lo siguiente es directamente incorrecto:

```

boolean encontrado = false;
for (int i = 0; i < arr.length; i++) {
    if (arr[i] == 0) {
        encontrado = true;
    }
    else {
        encontrado = false
    }
}
return encontrado;

```

No es recomendable el uso de **break**, **continue**, o **return** dentro de bucles porque entonces la condición de salida del bucle no es la negada de la condición de continuación. Además dificulta la legibilidad del código.

```

for (int i = 0; i < arr.length; i++) {
    if (arr[i] == 0) {
        return true;
    }
}
return false;

```

Puede no respetarse la invariante del bucle (la propiedad que se cumple en cada paso del bucle) y hay que leer todo el bloque para determinar todas las condiciones de salida que no están en la condición de terminación. Se complica la comprensión y el mantenimiento del código. En este ejemplo el código del bucle es corto pero hay bucles muy grandes. En la práctica no tiene por qué ser necesario leer el bloque. Dentro del bucle pueden modificarse muchas variables del programa, pero una vez terminado el bucle, se puede escribir junto con la negada de la condición de continuación la condición que se cumple sobre todas las variables afectadas por el bucle. El programador no necesita saber qué hace el bucle sino qué condiciones se cumplen antes y después.

En general, podríamos decir que la **estructura “típica”** de un método podría ser algo así:

```

// método que busca m en array
// PRE: m > 0 (por poner alguna)
public boolean metodo (int m, int arr[]) {
    if (m < 0) { // Evaluamos la/s precondition/es del método
        throw new IllegalArgumentException ("m no es positivo");
    }

    if (arr == null || arr.length == 0) { // Caso/s frontera
        return false;
    }

    int i = 0;
    while (i < arr.length && arr[i] != m) {
        i++;
    }

    return i < arr.length;
}

```

Paso de parámetros

El paso de parámetros en **Java** siempre es **por valor**, es decir, se realiza una copia del valor de la variable que se pasa como parámetro por cada una de las llamadas a un método. El comportamiento es diferente dependiendo del tipo de la variable.

El siguiente fragmento de código intercambia el valor de dos variables utilizando una variable auxiliar:

```

public static void main(String [] args) {
    int x = 5;
    int y = 4;

    int tmp = x;    // código del intercambio ("swapping")
    x = y;
    y = tmp;
}

```

El siguiente método que pretende reutilizar el código del intercambio no funciona:

```

public static void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

public static void main(String [] args) {
    int x = 5;
    int y = 4;

    this.swap(x,y); // no funciona

    int z = 7;
    int k = 8;

    this.swap(z,k); // no funciona
}

```

Sin embargo, este método sí que intercambia correctamente los valores de `array[0]` y `array[1]`:

```

public static void swap(int array[]) {
    int tmp = array[0];
    array[0] = array[1];
    array[1] = tmp;
}

```

Esto se debe a que `swap` realmente no recibe una copia del array completo, sino que lo que está recibiendo es una copia del *puntero* al array, con lo que el contenido del array se modifica correctamente.

Esto puede producir algunos problemas típicos:

```

public static void creaObjeto(Object o) {
    o = new Object ();
}

void main (String [] args) {
    Object o = null;
    creaObjeto(o);
    // o apunta al nuevo objeto?
}

```

Clases, objetos y variables

Las clases definen los métodos y atributos ("fields") en el argot de Java, que tendrán los objetos (instancias) de la clase. Por ejemplo:

```

public class Vehiculo {
    protected int ancho;
    protected int largo;
    protected int alto;

    /** Constructor por defecto */
    public Vehiculo() { ancho = largo = alto = 0; }

    public Vehiculo(int a, int l, int h) {
        ancho = a;
        largo = l;
    }
}

```

```

        alto    = h;
    }

    /** Constructor de copia */
    public Vehiculo(Vehiculo v) {
        ancho = v.ancho;
        largo = v.largo;
        alto  = v.alto;
    }

    /** ``Getter`` */
    public int getDimension() { return ancho * largo * alto; }

    /** ``Setter`` */
    public void setAncho(int a) { ancho = a; }

    public boolean equals(Object o) {
        if (o == this) return true;
        if (o instanceof Vehiculo) {
            Vehiculo v = (Vehiculo) o; // downcasting
            return ancho == v.ancho && largo == v.largo && alto == v.alto;
        } else return false; // tambien si o == null
    }
}

```

Los *objetos* (instancias) se crean en tiempo de ejecución con **new**. Las *variables* de tipo clase referencian objetos. Se puede cambiar el objeto al que referencia una variable mediante una asignación de ésta a otro objeto. El recolector de basura recogerá los objetos que no son referenciados por ninguna variable o posición de array, etc.

Ejercicio 1.4 Dibuja las variables, los objetos, y las flechas (referencias) entre variables y objetos que resultan de ejecutar el siguiente código:

```

public static void main(String [] args) {
    Vehiculo v1 = new Vehiculo(10,10,10);
    Vehiculo v2 = new Vehiculo(20,20,20);
    Vehiculo v3;

    v3 = v1;
    v1 = v2;
    v3 = new Vehiculo(10,10,10);
}

```

Ahora vamo a ver un ejemplo con objetos, variables, referencias, e igualdad:

```

public static void main(String [] args) {
    Vehiculo v1 = new Vehiculo(10,10,10);
    Vehiculo v2 = v1;

    if (v1.equals(v2)) // cierto
        System.out.println("Los Vehiculos son iguales usando equals");

    if (v1 == v2) // cierto
        System.out.println("Las variables referencian el mismo objeto");

    v2 = new Vehiculo(10,10,10);

    if (v1.equals(v2)) // cierto
        System.out.println("Los Vehiculos son iguales usando equals");

    if (v1 == v2) // falso
        System.out.println("Las variables referencian el mismo objeto");
}

```

En el argot técnico, se habla de **identidad** y **estado** de un objeto. Las variables referencian la identidad (generalmente la dirección de memoria donde está el objeto) mientras que los valores de los atributos (contenido) del objeto determinan su estado.

El método `equals` se usa para comparar el estado mientras que el operador `==` se usa para comparar identidades. Para expresiones de tipo primitivo sólo puede usarse `==`. Véase <http://www.javapractices.com/topic/TopicAction.do?Id=17> saber más sobre la dificultad de programar el método `equals`. El siguiente fragmento de código intercambia el valor de dos variables referencia utilizando una variable auxiliar:

```
Vehiculo x = new Vehiculo(10,10,10);
Vehiculo y = new Vehiculo(20,20,20);

Vehiculo tmp = x;    // intercambio (''swapping'')
x = y;
y = tmp;
```

El siguiente método que pretende reutilizar el código del intercambio no funciona:

```
public static void swap(Vehiculo a, Vehiculo b) {
    Vehiculo tmp = a;
    a = b;
    b = tmp;
}

public static void main(String [] args) {
    Vehiculo x = new Vehiculo(10,10,10);
    Vehiculo y = new Vehiculo(20,20,20);

    this.swap(x,y); // no funciona
}
```

Clases envoltorio

Clases envoltorio ("wrapper classes") `'Integer'`, `'Character'`, `'Boolean'`, etc, y sus conversiones automáticas ("autoboxing", "boxing", "unboxing"). Mira qué curioso lo que pasa en este `main`:

```
public static void main(String[] args) {
    Integer a, b;

    a = b = 3;
    System.out.println("Case 1: `` + (a == b ? ``equal`` : ``different``));

    a = 3;
    b = 3;
    System.out.println("Case 2: `` + (a == b ? ``equal`` : ``different``));

    a = b = 300;
    System.out.println("Case 3: `` + (a == b ? ``equal`` : ``different``));

    a = 300;
    b = 300;
    System.out.println("Case 4: `` + (a == b ? ``equal`` : ``different``));
}
```

Salida por pantalla:

```
Case 1: equal
Case 2: equal
Case 3: equal
Case 4: different
```

El puntero `this`

La palabra reservada `this` es una constante, no puede modificarse. Para recordar qué es `this` analiza éstos fragmentos de código, algunos de los cuales compilan pero son incorrectos y otros directamente ni compilan.

Ejercicio 1.5 ¿Cuál de estos fragmentos son incorrectos?

```

public class C {
    public int x;

    public C(int x) {
        this.x = x;
    }
}

public class C {
    public int x;

    public C(int y) {
        x = y;
    }
}

public class C {
    public int x;

    public C(int x) {
        x = x;
    }
}

public class C {
    public int x;

    public C(int x) {
        this.x = this.x;
    }
}

public class C {
    public int x;

    public C(C c2) {
        this = c2;
    }
}

```

Composición: relación “tiene-un”

```

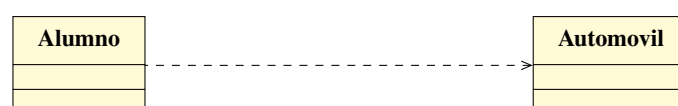
public class Automovil { ... }

public class Alumno {
    private Automovil a;
    ...
}

```

Un objeto de clase 'Alumno' “tiene-un” objeto de clase `Automovil` como atributo. En el ejemplo anterior el atributo es privado, únicamente se puede acceder a él dentro de la clase `Alumno`.

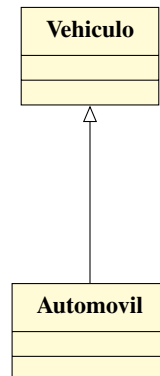
Diagrama de clases que usaremos en éste guión:

**Herencia (“inheritance”): relación “es-un”**

Veamos un ejemplo típico de herencia

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
```

Diagrama de clases UML:



Un objeto de clase `Automovil` también “es-un” objeto de clase `Vehiculo`. La palabra clave “extends” dice que la clase `Vehiculo` extiende la clase `Vehiculo`, la hace más específica aportando nuevos atributos o métodos (o también sobrescribiendo métodos).

La herencia permite reusar código: un objeto de la clase `Vehiculo` tiene todos los atributos y métodos de la clase `Vehiculo`.

Hay muchos aspectos técnicos: subclase vs subtipo, leyes semánticas, monotonía, sobrescritura, reemplazo vs refinamiento, Principio de Sustitución de Liskov, co-varianza y contra-varianza, etc. Para más información, leed los libros de [Budd:1991] y [Meyer:1988].

Upcasting

El upcasting es un mecanismo *de compilación* para convertir el tipo (“cast”) yendo hacia arriba (“up”) en la jerarquía de clases. La conversión la hace el compilador automáticamente. No es necesario que el programador escriba una expresión de “casting”. Más concretamente:

- Se puede asignar un objeto de una subclase a una variable de tipo superclase:

```
Vehiculo v = new Automovil(); // upcasting
```

El tipo de una variable en tiempo de compilación es aquel con el que la variable ha sido declarada. En este ejemplo el tipo de `v` es `Vehiculo`.

Pero en tiempo de ejecución, al crearse el objeto con `new` la variable `v` referencia un objeto de clase `Vehiculo`.

La línea compila porque el compilador ha realizado la **conversión de tipo** de forma automática en la asignación: a la derecha del `=` se tiene una expresión que evalúa a un objeto de tipo `Vehiculo` y a la izquierda se tiene una variable de tipo `Vehiculo`.

Para el compilador, la variable `v` es y será de tipo `Vehiculo`. El **objeto no se ha convertido**. De hecho, no existe pues se crea en ejecución.

En tiempo de ejecución todo objeto almacena internamente una representación de su tipo.

- La variable `v` es una *variable polimórfica*: puede referenciar objetos de clase `Vehiculo` así como objetos de cualquiera de las subclases de `Vehiculo`.
- También se puede hacer “upcasting” en el paso de parámetros. Por ejemplo, el siguiente método puede invocarse pasándole como parámetro un objeto de clase `Vehiculo`:

```
public void método(Vehiculo v) { ... }
```

- Un método puede devolver un objeto de la subclase cuando su tipo de retorno es la superclase:

```
public Vehiculo método() { return new Automovil(); }
```

- Se pueden combinar éstos dos aspectos. Supongamos que disponemos de los siguientes métodos:

```
public Automovil m1() { return new Automovil(); }
public Vehiculo m2() { return new Automovil(); }
public void m3(Vehiculo v) { ... }
```


Los siguientes son todos “upcastings” válidos (asumiendo que ‘o’ es un objeto de la clase donde se definen los métodos ‘m1’, ‘m2’ y ‘m3’ anteriores):

```
Vehiculo v1 = o.m1();
Vehiculo v2 = o.m2();
o.m3(new Automovil());
```

Y ¿para qué sirve el “upcasting”? Tiene varios usos, pero el uso principal tiene que ver con el enlazado dinámico.

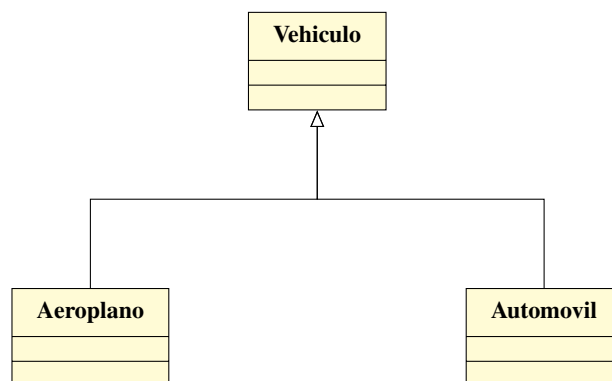
Upcasting

El upcasting es un Mecanismo *de compilación* para convertir el tipo (“cast”) yendo hacia abajo (“down”) en la jerarquía de clases. La conversión debe especificarla explícitamente el programador anotando el tipo deseado entre paréntesis (una expresión de “casting”).

Supongamos que tenemos:

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
public class Aeroplano extends Vehiculo { ... }
```

Diagrama de clases:



Veamos:

```
Vehiculo v = new Automovil(); // upcasting

Automovil a1 = v; // no compila
```

Para convertir el tipo en tiempo de compilación deberíamos tener la garantía de que **en tiempo de ejecución** la variable ‘v’ referencia un objeto de clase `Vehiculo` y no de clase ‘Aeroplano’.

En este ejemplo se da el caso, pero en general en **tiempo de compilación** el compilador no puede determinar el tipo del objeto al que referenciará una variable **en tiempo de ejecución**. Más concretamente, un compilador no puede para todos los casos determinar los valores que tendrán las variables en tiempo de ejecución para todos los posibles programas. ¡Para eso hay que ejecutar los programas! Esto es así para variables de cualquier tipo.

Es responsabilidad del programador indicar la conversión explícitamente. El “downcasting” es decirle al compilador “tranquilo, sé lo que hago”.

Si el programador se equivoca, puesto que el compilador no puede ayudarlo, descubrirá su error en tiempo de ejecución.

Algunos ejemplos de downcasting:

```
Vehiculo v = new Automovil(); // upcasting

Automovil a2 = (Automovil) v; // downcasting, compila y correcto

Aeroplano p2 = (Aeroplano) v; // downcasting, compila pero incorrecto
```

Al ejecutar el programa el último “casting” lanzará la excepción ‘`ClassCastException`’.

Java ofrece **`instanceof`** al programador para averiguar el “Run-Time Type Information” (RTTI), es decir, para preguntar el tipo en tiempo de ejecución del objeto. Como hemos dicho antes, los objetos almacenan una representación de su tipo.

```
Vehiculo v = new Automovil(); // upcasting

if (v instanceof Automovil)

    Automovil a = (Automovil) v;

else if (v instanceof Aeroplano)

    Aeroplano p = (Aeroplano) v;
```

Pero **instanceof** tiene dos limitaciones:

1. No fuerza a que la pregunta y la acción estén coordinadas. El siguiente código erróneo compila con consecuencias desastrosas al ejecutar:

```
Vehiculo v = new Automovil(); // upcasting

if (v instanceof Automovil)
    Aeroplano p = (Aeroplano) v;
```

En otros lenguajes sí se fuerza. Por ejemplo C++ ofrece un método genérico “dynamic cast” que recibe una clase y un objeto y pregunta si el objeto es una instancia de la clase. En caso positivo devuelve el objeto tras un casting a dicha clase. En caso negativo, devuelve null.

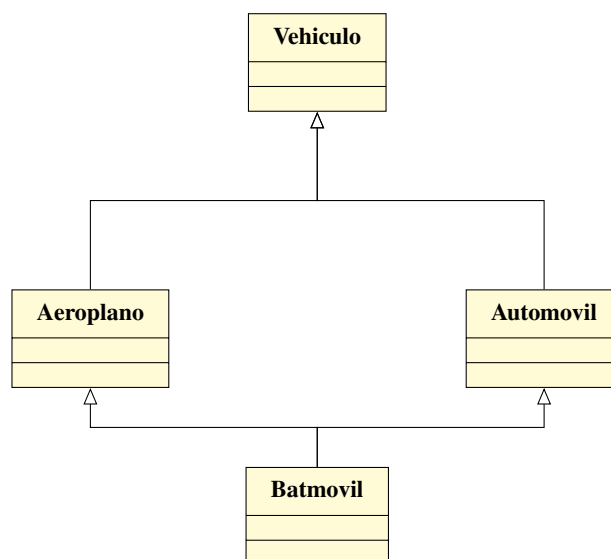
2. Se pueden crear modularmente (sin tener que recompilar las clases ya compiladas) nuevas subclases de la superclase. Pero el código donde está “instanceof” no puede extenderse modularmente para tener en cuenta esos nuevos casos. Por cada nueva clase hay que editar el código, añadir nuevas ramas “if” y recompilar. Este problema se llama “Expression Problem” y hay soluciones, algunas muy buenas en otros lenguajes.

Herencia múltiple y “Diamond of Death”

Supongamos que tenemos la siguiente herencia:

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
public class Aeroplano extends Vehiculo { ... }
public class Batmovil extends Automovil, Aeroplano { ... }
// La ultima linea no compila en Java
```

Se produce el “Diamond of Death”:



Las clases `Vehiculo` y `Aeroplano` heredarían los atributos y métodos públicos y protegidos de `Vehiculo`. ¿De quién los hereda `Batmóvil` el atributo `'ancho'` común a `Vehiculo`, `Vehiculo` y `Aeroplano`? Hay una ambigüedad que debe resolverse.

Java no permite herencia múltiple Solución de Java: se introducen los **interfaces** para permitir herencia múltiple. Los interfaces no definen atributos ni código, únicamente nombres de métodos. Los veremos posteriormente.

Esta es la motivación principal de los interfaces, y no la separación entre especificación e implementación para TADs, aunque hay una relación débil, como veremos más adelante.

Enlazado dinámico y sobrescritura

La segunda utilidad es poder invocar un método sobre una variable polimórfica de forma que el método seleccionado dependa del tipo del objeto que referencia la variable en tiempo de ejecución. Un ejemplo:

```
public static void main (String [] args) {
    Vehiculo v[] = new Vehiculo[3];
    v[0] = new Automovil();
    v[1] = new Aeroplano();
    v[2] = new Barco();

    for (int i = 0; i < v.length; i++)
        v[i].display();    // dibuja el objeto en pantalla
}
```

El objetivo es poder utilizar una variable de tipo superclase que referencie objetos de subclases (por tanto una variable polimórfica) e invocar un método *sobrescrito* en las subclases de forma que el comportamiento se adecúe a lo esperado para las subclases.

El ejemplo asume que el método `display` está definido en la clase 'Vehiculo'. Supongamos que lo que dibuja es un cubo del tamaño de las dimensiones dadas por los atributos 'ancho', etc.

Si las clases `Automovil`, `Aeroplano` y `Barco` no sobrescriben el método `display` para dibujar respectivamente un Automovil, un aeroplano y un barco, entonces en tiempo de ejecución se invocará el método `display` heredado de `Vehiculo`. Es decir, se dibujarán tres cubos. Pero si todas sobrescriben adecuadamente el método `display` entonces se dibujará respectivamente un Automovil, un aeroplano y un barco.

Si se añaden nuevas subclases a `Vehiculo` entonces se podrían almacenar en el vector objetos de las nuevas subclases. Pero habría que modificar las inicializaciones para almacenar los objetos y habría que recompilar. Como en el caso de **instanceof**, se pueden añadir nuevas clases de forma modular pero el código ya escrito no puede extenderse de forma modular—sin utilizar trucos, claro.

Terminología

- **Sobrescritura** (*overriding*): Cuando una subclase redefine un método o atributo de una superclase. (Debe usarse la anotación `@Override` de Java sobre las cabeceras de los atributos y métodos sobrescritos.)
- **Sobrecarga**: Hay que diferenciar la sobrescritura (*overriding*) de la sobrecarga ("overloading"). No son lo mismo y es fundamental comprender la diferencia entre ambas. La sobrecarga implica tener más de un método con el mismo nombre, pero con distintos parámetros y para esto no es necesario que haya herencia.
- **Enlazado dinámico** (*dynamic dispatching*): Dada una variable que referencia un objeto, a través de la cual se accede a un atributo público o se invoca un método del objeto, qué atributo se accede y qué método se invoca está determinado por el tipo **en tiempo de ejecución** del objeto al que referencia la variable, no por el tipo con que ésta se declara en tiempo de compilación. Esto es así para atributos o métodos sobrescritos. Es decir, en

```
Vehiculo v = new Automovil();
v.display();
```

El método 'display' invocado será el de la clase `Vehiculo` si dicho método se hereda de 'Vehiculo' y se sobrescribe en `Vehiculo`.

Este concepto no tiene nada que ver con otro de similar nombre: el enlazado dinámico de librerías (*dynamic linking*).

- **Enlazado dinámico** no es la traducción más precisa del Inglés. Por un lado tenemos *dynamic dispatching* (elegir la implementación de un método en tiempo de ejecución) y por otro *dynamic binding* (también llamado *late binding*: asociar un nombre en tiempo de ejecución). "Bind" se suele traducir por "enlazar o asociar". En Java, la asociación del nombre del método se hace en tiempo de compilación pero la elección de la implementación del método se hace en tiempo de ejecución. Más concretamente, la asociación de un método a una variable polimórfica se fija en tiempo de compilación. Por ejemplo, `v.display()` compila si el método `display` está definido en la clase `Vehiculo`. En tiempo de ejecución se elige la implementación del método 'display' de la clase del objeto al que referencia 'v'.

Enlazado dinámico

Notad que no hay enlazado dinámico sin sobrescritura

```
public class Vehiculo {
    public void display() { ... }
}
public class Automovil extends Vehiculo {
    public int matricula() { ... } // método que no esta en 'Vehiculo'
}

Vehiculo v = new Automovil(); // upcasting
v.display(); // compila, invoca el método 'display' heredado.
v.matricula(); // no compila: Vehiculo no tiene un método 'matricula'

Automovil a = new Automovil();
a.display(); // compila, invoca el método 'display' heredado.
a.matricula(); // compila, invoca el método 'matricula' de Vehiculo
```

En `v.display()` la variable `'v'` almacena un `Vehiculo`. Por tanto se invoca al objeto `Automovil` el método heredado de `'Vehiculo'`. El comportamiento será el mismo que si en `'v'` hubiera un objeto de clase `Vehiculo`: se dibujará un cubo.

```
public class Vehiculo {
    public void display() { ... }
}
public class Automovil extends Vehiculo {
    @Override
    public void display() { ... } // sobrescribe el de 'Vehiculo'

    public int matricula() { ... } // método que no esta en 'Vehiculo'
}
```

Suponemos que el método `display` de `Automovil` se sobrescribe para que dibuje un `Automovil`.

```
Vehiculo v = new Automovil(); // upcasting
v.display(); // compila, invoca el método sobrescrito, no el heredado.
v.matricula(); // no compila, 'Vehiculo' no tiene un método 'matricula'

Automovil a = new Automovil();
a.display(); // compila, invoca el método sobrescrito, no el heredado.
a.matricula(); // compila, invoca el método 'matricula' de Automovil
```

En `v.display()` la variable `'v'` almacena un `Automovil`. Por tanto se invoca al objeto `Automovil` su método `display` sobrescrito. La invocación `v.display()` dibujará el `Automovil`.

El comportamiento no es el mismo que si `'v'` referenciara un `Vehiculo` pues en este caso se dibujaría un cubo. El comportamiento es el mismo que el de la invocación `a.display()` posterior.

Repetimos: La idea es utilizar una variable de tipo superclase a la que se asignan objetos de subclases. A esa variable se le invoca un método común sobrescrito en las subclases.

¿Se podría conseguir lo mismo usando “downcastings”?

Uno puede preguntarse si con “downcastings” sería suficiente y no haría falta el enlazado dinámico.

```
public class Vehiculo {
    public void display() { ... }
}
public class Automovil extends Vehiculo {
    @Override
    public void display() { ... }
}
public class Aeroplano extends Vehiculo {
    @Override
    public void display() { ... }
}
```

La sobrescritura es necesaria para que el compilador acepte invocar el método 'display' a una variable de tipo Vehiculo.

Podríamos pensar que con “downcastings” es suficiente:

```
public static void main (String [] args) {
    Vehiculo v[] = new Vehiculo[2];
    v[0] = new Automovil();
    v[1] = new Aeroplano();

    ((Automovil)v[0]).display(); // dibuja un Automovil
    ((Aeroplano)v[1]).display(); // dibuja un aeroplano
}
```

El problema es que esto no nos permite escribir código uniforme. Por ejemplo no podemos invocar display en un bucle:

```
public static void main (String [] args) {
    Vehiculo v[] = new Vehiculo[2];
    v[0] = new Automovil();
    v[1] = new Aeroplano();

    for (int i = 0; i < v.length; i++)
        ((QUE_CASTING_PONGO_AQUI?)v[i]).display();
}
```

El enlazado dinámico permite no tener que utilizar “downcastings”.

La herencia y el enlazado dinámico tienen algunos peligros. El siguiente ejemplo ilustra el comportamiento del enlazado dinámico que puede llevar a confusión si no se conoce. Se tiene una clase 'Padre' con un método 'ricosuave' que invoca los métodos 'rico' y 'suave' de dicha clase. (Los alumnos nacidos después de 1990 probablemente no reconocerán la alusión a la “curiosa” canción del peculiar rapero Gerardo Mejía.) La clase 'Hijo' extiende la clase padre sobrescribiendo todos los métodos, refinándolos. La clase 'Main' crea objetos de clase 'Padre' y de clase 'Hijo'. La variable 'phijo' es de tipo 'Padre' pero referencia en ejecución un objeto de clase 'Hijo'.

```
public class Padre {

    public void rico() {
        System.out.print("Rico. ");
    }

    public void suave() {
        System.out.print("Suave.");
    }

    public void ricosuave() {
        this.rico();
        this.suave();
    }
}

public class Hijo extends Padre {

    @Override
    public void rico() {
        System.out.print("Muy ");
        super.rico();
    }

    @Override
    public void suave() {
        System.out.print("Muy ");
        super.suave();
    }

    @Override
    public void ricosuave() {
```

```

        System.out.print('Muy ');
        super.ricosuave();
    }
}

public class Main {
    public static void main(String [] args) {
        Padre padre = new Padre();
        padre.ricosuave();
        System.out.println();

        Hijo hijo = new Hijo();
        hijo.ricosuave();
        System.out.println();

        Padre phiho = new Hijo();
        phiho.ricosuave();
        System.out.println();
    }
}

```

La salida por pantalla **no** es:

```

Rico. Suave.
Muy Rico. Suave.
Muy Rico. Suave.

```

La salida por pantalla es:

```

Rico. Suave.
Muy Muy Rico. Muy Suave.
Muy Muy Rico. Muy Suave.

```

El comportamiento para 'hijo' y 'phiho' es el mismo, como cabe esperar por el enlazado dinámico.

Esta es la traza de ejecución de la invocación hijo.ricosuave():

- se muestra "Muy" por pantalla. - se invoca super.ricosuave() - se invoca this.rico() pero 'this' es el objeto al que referencia 'hijo' es decir, se invoca el método 'rico' de 'Hijo'. - se muestra "Muy" por pantalla. - se invoca super.rico() - se muestra "Rico" por pantalla. - se invoca this.suave() pero 'this' es el objeto al que referencia 'hijo' es decir, se invoca el método 'suave' de 'Hijo'. - se muestra "Muy" por pantalla. - se invoca super.suave() - se muestra "Suave" por pantalla.

Herencia y sobrecarga

Decimos que un atributo está sobrecargado cuando aparece declarado con distinto tipo en la misma clase. En esta asignatura no usaremos atributos sobrecargados. Un método está sobrecargado cuando aparece declarado en la misma clase con algún tipo distinto en sus parámetros. Por ejemplo:

```

public class Vehiculo {
    ...
    public int display() { ... }
    public void display(Window w) { ... }
}

```

La sobrecarga de atributos y métodos se resuelve en tiempo de compilación. Para los métodos, la sobrecarga se resuelve en función del tipo de los parámetros del método.

Cuando el programador escribe 'v.display(w)' con 'v' de tipo Vehiculo o subclase de Vehiculo entonces el compilador determina que tiene que invocarse el segundo método si 'w' es de clase 'Window' o de cualquier subclase de 'Window'.

Si no puede resolverse la sobrecarga porque hay ambigüedad, entonces el el compilador indica el problema y el programa no compila correctamente.

La sobrescritura es una especie de sobrecarga que se resuelve trivialmente mediante el ámbito y el enlazado dinámico.

```

public class Vehiculo {
    public void display(Window w) { ... }
}

```

```
public class Automovil extends Vehiculo {
    @Override
    public void display(Window w) { ... }
    public int display() { ... }
}
```

Cuando el programador escribe `v.display(w)`, qué método se invoca depende del tipo del objeto al que referencia `v` en tiempo de ejecución.

Ejercicio 1.6 ¿Qué pasa si `Automovil` no sobrescribe el método `void display(Window w)`? ■

Ejercicio 1.7 ¿Qué pasa con la invocación `v.display()` si `v` es de tipo `Vehiculo` pero referencia un objeto `Automovil`? ■

La interacción entre sobrecarga y herencia complica la resolución de la sobrecarga:

```
public class Vehiculo {
    public int display(Vehiculo v) { ... }
    public void display(Automovil a) { ... }
}
```

¿Cómo se resuelve la sobrecarga en los siguientes casos (suponiendo que `Automovil` no sobrescribe ni sobrecarga `display`)?

```
Vehiculo v = new Vehiculo();
Automovil a = new Automovil();
v.display(v);
v.display(a);
```

```
Vehiculo v = new Automovil();
Automovil a = new Automovil();
v.display(v);
v.display(a);
```

■ **Ejemplo 1.1** Ejemplo de interacción entre sobrecarga y sobrescritura:

```
public class Vehiculo {
    public void chocar(Vehiculo v) { ... }
}

public class Automovil extends Vehiculo {
    public void chocar(Automovil a) { ... }
}

public class Main {
    public static void main (String args []) {

        Vehiculo v = new Automovil(); /* upcasting */
        Automovil a = new Automovil();

        v.chocar(a); // compila y
                    // se invoca chocar heredado de \linline{Vehiculo}!
    }
}
```

El método `'chocar'` no está `'sobrescrito'` en `Automovil`, está sobrecargado, pues `Automovil` hereda `'chocar'` de `Vehiculo` y tiene por tanto dos métodos `chocar` que toman parámetros de distinto tipo. ■

La sobrecarga se resuelve en tiempo de compilación: `v` es de tipo `Vehiculo` (aunque referencia un objeto de tipo `Automovil`). El compilador determina que se invoca el `'chocar'` de `Vehiculo`, que en tiempo de ejecución es el `'chocar'` que el objeto `Automovil` hereda de `Vehiculo`.

Ejercicio 1.8 Probar y explicar lo que sucede al cambiar el código del 'main' anterior según distintas combinaciones:

```
Vehiculo v = new Vehiculo();
Automovil a = new Automovil();
v.chocar(a);
a.chocar(v);

Vehiculo v1 = new Vehiculo();
Vehiculo v2 = new Automovil();
v1.chocar(v2);
v2.chocar(v1);

Vehiculo v1 = new Automovil();
Vehiculo v2 = new Automovil();
v1.chocar(v2);
v2.chocar(v1);

Automovil a = new Automovil();
Vehiculo v = new Automovil();
v.chocar(a);
a.chocar(v);

Automovil a1 = new Automovil();
Automovil a2 = new Automovil();
a1.chocar(a2);
a2.chocar(a1);

Vehiculo v1 = new Automovil();
Vehiculo v2 = new Automovil();
v1.chocar((Automovil)v2);
v2.chocar(v1);
```

Hay lenguajes en los que la herencia lleva pareja la “covarianza” del tipo de los parámetros de los métodos, que van hacia abajo en la jerarquía de interfaces/clases cuando éstos van hacia abajo. En Java no hay covarianza, pero si la hubiera:

```
public class Vehiculo {
    public void chocar(Vehiculo v) { ... }
}

public class Automovil extends Vehiculo {
    /* hereda 'void chocar(Automovil v)'
     * con el tipo del parametro 'v' yendo hacia abajo en la jerarquia
     */

    public void chocar(Automovil v) { ... }
    /* ya no seria una sobrecarga sino una sobrescritura */
}

public class Main {
    public static void main (String args []) {

        Vehiculo v = new Automovil(); /* upcasting */
        Automovil a = new Automovil();

        v.chocar(a); // compila, enlazado dinamico, invoca el sobrescrito
    }
}
```


Ejercicio 1.9 ¿Permite Java la covarianza en el tipo del valor de retorno de un método?

```
public class Vehiculo {
    public Vehiculo m() { ... }
}

public class Automovil {
    public Automovil m() { ... }
}
```

El método 'm' devuelve un objeto de tipo diferente, ¿se trata de sobrecarga, sobreescritura, puede el "upcasting" tener algo que ver aquí?

Pista: ¿se toma en cuenta el tipo del valor de retorno del método en la resolución de la sobrecarga? ■

1.2 Introducción a los TADs y los algoritmos

Empecemos con algunas definiciones:

- Tipo Abstracto de Datos (TAD): [Goodrich] "a systematic way of organizing and accessing data". Se trata de un concepto anterior e independiente del concepto de objeto.
- Algoritmos [Goodrich]: "a step-by-step procedure for performing some task in a finite amount of time". Ejemplos de algoritmos son las operaciones con o sobre TADs.
- Los lenguajes de programación ofrecen mecanismos (p.ej., interfaces, clases, objetos, herencia, genéricos, etc) para implementar TADs.

Vamos a ver un ejemplo motivador. Tenemos una clase para números complejos implementados mediante representación cartesiana.

```
public class ComplejoCart {
    private double real;
    private double imaginaria;

    public ComplejoCart(double re, double im) {
        this.real = re;
        this.imaginaria = im;
    }

    public ComplejoCart(double re) { // real a complejo
        this.real = re;
        imaginaria = 0.0;
    }

    public ComplejoCart(ComplejoCart c) { // constructor de copia
        real = c.getReal();
        imaginaria = c.getImaginaria();
    }

    // ``getters``

    public double getReal() { return real; }

    public double getImaginaria() { return imaginaria; }

    public double getModulo() {
        return ... // Aquí va una fórmula sobre 'real' e 'imaginaria'
                  // para obtener el módulo.
    }

    public double getArgumento() {
        return ... // Aquí va una fórmula sobre 'real' e 'imaginaria'
                  // para obtener el ángulo.
    }

    public ComplejoCart suma(ComplejoCart c) {
        return new ComplejoCart(this.real + c.getReal(), this.imaginaria + c.getImaginaria());
    }
}
```

```

    }

    public ComplejoCart resta(ComplejoCart c) {
        return new ComplejoCart(this.real - c.getReal(), this.imaginaria - c.getImaginaria());
    }

    public String toString() {
        return new String("'" + this.real + "', '" + this.imaginaria + "'");
    }

    public boolean equals(Object o) {
        if (o == this) return true;
        if (o instanceof ComplejoCart) {
            return this.real == c.getReal() && this.imaginaria == c.getImaginaria();
            ComplejoCart c = (ComplejoCart) o; // downcasting
        } else
            return false;
    }
}

```

Los métodos 'suma' y 'resta' no modifican el objeto sobre el que se invocan, devuelven un objeto nuevo. Esta es la forma natural de operar con valores matemáticos:

```

public static void main(String [] args) {
    ComplejoCart c1 = new ComplejoCart(1.0, 2.0);
    ComplejoCart c2 = new ComplejoCart(2.0, 3.0);

    ComplejoCart c3 = c1.suma(c2);

    System.out.println(c3.toString());
}

```

Acorde con la implementación del método 'suma', 'c1' no queda modificado al invocarle el método 'suma' sino que el método devuelve un nuevo objeto de clase `ComplejoCart` con la suma de los atributos de 'c1' y 'c2'. En el ejemplo hemos "guardado" (la referencia a) dicho objeto en otra variable 'c3'.

Observad además que no hay métodos "setters" para modificar el estado (los atributos) del objeto.

Los objetos cuyo estado no se modifica se llaman **objetos inmutables**. Las entidades matemáticas como los números se suelen implementar como objetos inmutables. Leed la discusión en los libros [Budd:1991], [Meyer:1988] y [Bloch:2008] sobre las ventajas e inconvenientes de los mismos. Otros ejemplos de objetos inmutables son los de clase envoltorio (Integer, Boolean, etc.) y los de clase 'String'.

Los TADs no tienen por qué consistir de objetos inmutables. En esta sección estamos usando un ejemplo con objetos inmutables por comenzar por algo sencillo, pero el concepto de TAD es independiente de si los objetos que lo implementan son inmutables o no.

Hemos usado los "getters" en 'c.getReal()' y 'c.getImaginaria()' en el código, pero podríamos haber accedido a los atributos del objeto referenciado por 'c' directamente pues es de clase `ComplejoCart`:

```

public ComplejoCart suma(ComplejoCart c) {
    return new ComplejoCart(this.re + c.re, this.im + c.im);
}

```

Problemas:

El código fuente revela la implementación. El cliente de la clase (en este ejemplo, el programador del método 'main') no está interesado en conocer los detalles de implementación sino en crear objetos `ComplejoCart` e invocarles métodos. Si le damos la clase pre-compilada en un ejecutable entonces el cliente no sabe qué métodos tiene que usar porque están en código binario. Habría que ofrecerle algún tipo de documentación (p.ej., Javadoc) aparte.

Si deseamos cambiar a una implementación con representación polar de complejos mediante módulo y ángulo entonces tenemos que reescribir la clase entera, o definir otra `ComplejoPolar`. Este sería el esquema de la implementación polar:

```

public class ComplejoPolar {
    private double modulo;
    private double argumento;
}

```

```

public ComplejoPolar(double mod, double ang) {
    this.modulo = mod;
    this.argumento = ang;
}

... // aquí van más constructores

public double getReal() { mod * Math.cos(ang) ; }
public double getImaginaria() { mod * Math.sin(ang) ; }
public double getModulo() { return mod; }
public double getArgumento() { return ang; }

public ComplejoPolar suma(ComplejoPolar c) {
    // la suma tiene un código diferente también.
}

... // etc
}

```

¡Cambia toda la implementación! Sin embargo el “cliente” de la clase (el código de `'main'`) cambiaría ligeramente:

```

public static void main(String [] args) {
    ComplejoPolar c1 = new ComplejoPolar(...); // nuevos constructores
    ComplejoPolar c2 = new ComplejoPolar(...);

    ComplejoPolar c3 = c1.suma(c2);

    System.out.println(c3.toString());
}

```

El cambio sería todavía menor si cada clase ofreciera métodos para transformar un cartesiano en polar y viceversa.

Ejercicio 1.10 ¿Ves en este ejemplo la desventaja de tener en Java los nombres de los constructores de una clase sobrecargados?

No se pueden combinar implementaciones diferentes. Sería cómodo poder sumar un objeto `ComplejoPolar` a un objeto `'ComplejoCartesiano'` pues ambos son números complejos aunque cambie la representación.

Clases, interfaces, herencia y TADs

En Java un TAD consiste en una jerarquía de interfaces y clases. En un TAD hecho en Java, los interfaces especifican (“qué”) y las clases implementan (“cómo”).

Veamos un ejemplo de interfaz para definir un número complejo:

```

public interface Complejo {

    /** Devuelve la parte real */
    public double getReal();

    /** Devuelve la parte imaginaria */
    public double getImaginaria();

    /** Devuelve el módulo */
    public double getModulo();

    /** Devuelve el argumento */
    public double getArgumento();

    /** Devuelve un nuevo Complejo haciendo la suma con 'c' */
    public Complejo suma(Complejo c);

    /** Devuelve un nuevo Complejo haciendo la resta con 'c' */
    public Complejo resta(Complejo c);
}

```

```
}
```

La semántica (es decir, la descripción precisa de qué debe hacer cada método) debe documentarse, bien informalmente mediante comentarios en lenguaje natural, en Javadoc, o mejor todavía, usando una notación formal.

El interfaz no incluye constructores, ni métodos heredados de 'Object' como 'toString', 'equals', etc. En Java estos métodos residen en las clases que implementan el interfaz.

Los interfaces de Java se introdujeron para la herencia múltiple y su uso como interfaces de TADs es ad-hoc. Pero es lo que hay.

Como hay dos formas de representar complejos, el interfaz ofrece “getters” para cada posible representación. Una razón de ser de los “getters” es ésta. No es necesario tener “getters” para clases con una representación única. Es válido acceder a los atributos directamente sin usar “getters” si la implementación de la clase no va a cambiar. (Sí, a pesar de lo que os hayan dicho por ahí.)

Las clases deben declarar que implementan el interfaz:

```
public class ComplejoCart implements Complejo {

    ... // los atributos, constructores, y 'getters' no cambian.

    public Complejo suma(Complejo c) { // Complejo es un interfaz
        return new ComplejoCart(this.re + c.getReal(), this.im + c.getImaginaria());
    }

    // ... tampoco cambia toString

    public boolean equals(Object o) {
        if (o == this) return true;
        if (o instanceof Complejo) {
            Complejo c = (Complejo) o; // downcasting al interfaz
            return this.re == c.getReal() && this.im == c.getImaginaria();
        } else return false;
    }
}
```

Los métodos 'suma' y 'resta' toman como parámetro un objeto **de cualquier clase que implementa el interfaz Complejo**. Por tanto, la implementación de dichos métodos debe tener esto en cuenta. Por ejemplo, el parámetro 'c' del método 'suma' o bien es null (no hay que olvidarse nunca de null, en nuestro ejemplo que se lance NullPointerException es razonable) o bien referencia un objeto de una clase que implementa el interfaz Complejo. Por otro lado, el método 'suma' devuelve un objeto de una clase que implementa el interfaz Complejo.

De igual manera, una declaración 'Complejo c1' lo que dice es que 'c1' almacenará o bien null o bien una referencia a un objeto de una clase que implementa el interfaz Complejo. Recordad:

```
Complejo c1 = new Complejo(); // suspenso
```

No tiene sentido y no compila.

Ventajas de usar “getters” sobre variables de tipo interfaz:

Gracias al uso de “getters”, tanto el constructor de copia, como el método suma y el método equals funcionan cuando 'c' es un objeto **de cualquier clase que implemente el interfaz**. Observad que en equals el “downcasting” se hace de Object a Complejo, que es un interfaz, no una clase.

Ejemplo de uso por el “cliente” del TAD, esta vez usando el interfaz y las dos implementaciones:

```
public static void main(String [] args) {
    Complejo c1 = new ComplejoCart(1.0, 2.0);
    Complejo c2 = new ComplejoPolar(8.4, 60.0);

    Complejo c3 = c1.suma(c2);

    System.out.println(c3.toString());

    System.out.println(c2.suma(c1).toString());
}
```

Las variables que utilizamos **son de tipo interfaz pero en tiempo de ejecución referencian objetos de clases que implementan el interfaz**.

La “librería” o paquete de números complejos consta de un interfaz y dos clases. El cliente de la librería sólo tiene que conocer cuáles son y qué hacen los métodos del interfaz, así como los nombres de las clases y los constructores. El resto de detalles de implementación no son necesarios.

Observad que `c1.suma(c2)` es un objeto de clase `ComplejoCart` mientras que `c2.suma(c1)` es un objeto de tipo `ComplejoPolar`.

Hay que tener cuidado al diseñar TADs con múltiples representaciones. Observad que en el `main` anterior `c1.suma(c2)` es un objeto de clase `ComplejoCart` mientras que `'c2.suma(c1)'` es un objeto de clase `ComplejoPolar`. Debe cumplirse que `'c1.equals(c2)'` da el mismo valor de verdad que `'c2.equals(c1)'` aunque tengan distinta representación. Aquí se reafirma el rol de los “getters” y que los cambios de representación sean biyectivos.

Acceso via “downcasting” a la clase que implementa el interfaz

En ocasiones es necesario acceder directamente a los atributos de la clase que implementa el interfaz en vez de utilizar los “getters” del interfaz. Explicaremos más adelante cuándo es necesario hacer ésto. De momento sólo mostramos cómo hacerlo.

Volvamos a la clase `ComplejoCart`. En la siguiente versión modificamos el código del método `'suma'` para que sólo sume con objetos de la clase `ComplejoCart`, accediendo directamente a los atributos de la clase vía “downcasting”:

```
public class ComplejoCart implements Complejo {
    ...

    public Complejo suma(Complejo c) throws IllegalArgumentException {
        if (c instanceof ComplejoCart)

            return new ComplejoCart( this.re + ((ComplejoCart)c).re
                                     , this.im + ((ComplejoCart)c).im );

        else

            throw new IllegalArgumentException();
    }
}
```

El acceso a la implementación puede complicar o imposibilitar al cliente el inter-operar con múltiples representaciones.

Resumen de ideas de este tema de introducción

- Un TAD consiste en uno o más interfaces y clases. Los interfaces especifican (“qué”) y las clases implementan (“cómo”).
- El “cliente” del TAD sólo tiene que conocer los interfaces, para qué sirven, qué hacen los métodos y con qué eficiencia. También debe conocer el nombre y los constructores de las clases que implementan los interfaces.
- El “cliente” NO tiene que conocer los atributos ni el código de los métodos de las clases. De hecho, puede recibir el código pre-compilado en un ejecutable.
- En lo posible, los interfaces y clases deben estar diseñados para poder combinar objetos de clases distintas. El código de las clases debe invocar sobre los objetos los métodos recogidos en el interfaz.
- Idealmente, el “implementador” de las clases puede cambiar los detalles de éstas sin que el código del “cliente” se vea afectado, ya que el cliente utiliza el interfaz.

Conceptos y terminología

- **Abstracción:** “Separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción.” (Diccionario de la Real Academia Española, 22ª edición)
- **Abstracción funcional:** atender a la función (funcionalidad, propósito, “el qué”) ignorando la estructura/representación interna (“el cómo”).
- La función se describe mediante un **interfaz** (p.ej., un interfaz de Java) y una especificación del comportamiento (semántica o contrato) que suele darse bien en lenguaje natural, bien en una notación semiformal (p.ej., Javadoc), bien en una notación formal (p.ej., lógica de orden superior).
- La estructura/representación interna se describe en una **implementación** (p.ej., una clase de Java) que realiza la funcionalidad acorde al contrato.

- **Encapsulación** (ocultación de información): sólo es necesario conocer la función, la estructura puede quedar oculta.
Ejemplo: el cliente de un TAD sólo necesita conocer los interfaces y los nombres y constructores de las clases que implementan los interfaces. Los detalles de implementación de las clases pueden quedar ocultos.
- **Independencia de la representación**: la misma función puede ser realizada por diferentes representaciones. Por ejemplo, el interfaz puede ser implementado por distintas clases.
- **Modularidad**: un módulo o cápsula realiza una funcionalidad. Debe tener **alta cohesión** (ser autocontenido) y **bajo acoplamiento** (no depender de cambios en otros módulos).
- **Abstracción de datos**: mecanismos para abstraer datos. Típicamente: tipos de datos, clases, paquetes, etc.
- **Abstracción de control**: macros, subrutinas, programación estructurada (condicionales, bucles, etc), procedimientos y funciones, objetos y métodos, mecanismos de sincronización y comunicación en concurrencia, etc.
- La definición matemática más aceptada de **TAD** es la de álgebra, en particular la de álgebra no libre: conjuntos de elementos que son manipulados en términos de operaciones sobre el conjunto las cuales cumplen leyes descritas mediante ecuaciones condicionales.
Por ejemplo, los enteros son un TAD. Son un conjunto de datos cuya representación interna en la máquina no conocemos y que tienen una serie de operaciones (suma, resta, división, etc.) donde cada una cumple propiedades (conmutativa, distributiva, etc.).

Idealmente, la independencia de la implementación implica bajo acoplamiento: se debería poder cambiar la implementación sin afectar el código cliente que usa los métodos del TAD. El interfaz sirve de “contrato” entre el cliente y el implementador del TAD.

Pero esto no es siempre así. Hay interfaces que por los parámetros que toman los métodos sugieren una implementación concreta y un cambio implica cambiar el interfaz: “interface coupling”.

Además, en Java no basta con conocer el interfaz, hay que conocer las clases y los constructores para crear los objetos. Si se cambia de implementación entonces hay que recompilar el código donde se crean los objetos. Ejemplo:

```
public class Main {  
    public static void main (String[] args) {  
        Complejo c = new ComplejoCart (3.32, 5.21);  
        Complejo s = c.suma(c);  
    }  
}
```

Para cambiar a representación polar hay que cambiar el constructor `ComplejoCart` por `ComplejoPolar` y recompilar.

Existen patrones de diseño para solucionar este problema, como el patrón factoría. La idea es de nuevo usar encapsulación. Se puede crear una clase intermedia que hace las veces de “interfaz” para los constructores.

Idealmente, los TADs tienen alta cohesión, bajo acoplamiento, y encajan con el diseño modular y el refinamiento progresivo (se pueden posponer decisiones de implementación durante el proceso de diseño). En la práctica, hay factores pragmáticos como la eficiencia que pueden requerir un cambio de implementación que afecta al interfaz o incluso requerir la sustitución de un TAD por otro. El software evoluciona. Hay métodos estadísticos y probabilísticos para la elección de la implementación adecuada de un TAD basándose en el análisis de las operaciones del TAD que el “cliente” utiliza en su código.

2. Listas Indexadas

Los TADs contenedores son aquellos que se diseñan para insertar (añadir, almacenar), buscar, y eliminar (borrar) datos (elementos) de forma eficiente. En el argot callejero: “meter”, “sacar”, “encontrar”. Una lista es un TAD contenedor que consiste en una secuencia lineal de elementos. Por ejemplo, una lista de la compra escrita con un editor de textos. Se puede insertar un elemento en cualquier posición de la lista, por ejemplo entre otros dos elementos. La lista no tiene por qué estar ordenada.

En programación, las listas se suelen escribir horizontalmente de izquierda a derecha. Los elementos están ordenados linealmente según su posición de izquierda a derecha: primer elemento, segundo, etc. La *posición* de un elemento cambia al insertar y al borrar elementos y se pueden insertar y borrar elementos en cualquier posición. La búsqueda de elementos suele ser secuencial o lineal: de izquierda a derecha o de derecha a izquierda. No hay acceso aleatorio como en los arrays.

De forma teórica, el tamaño de la lista no está acotado y podría aumentar indefinidamente. En la práctica el tamaño está acotado por la memoria disponible del computador. Puesto que el computador no tiene memoria infinita, podrá producirse una excepción de insuficiente memoria al invocar un método de inserción.

A continuación vamos a repasar el interfaz `IndexedList<E>` que se vio en la asignatura Programación II y que se trata de una lista que utiliza un índice numérico para el acceso a los elementos de la lista.

2.1 Interfaz `IndexedList`

```
public interface IndexedList<E> extends Iterable<E> {  
  
    public void add(int index, E e) throws IndexOutOfBoundsException;  
  
    public E get(int index) throws IndexOutOfBoundsException;  
  
    public boolean isEmpty();  
  
    public int size();  
  
    public E set(int index, E e) throws IndexOutOfBoundsException;  
  
    public int indexOf(E search);  
  
    public E removeElementAt(int i) throws IndexOutOfBoundsException;  
  
    public boolean remove(E element);  
  
    public Object [] toArray();  
}
```

El interfaz `IndexedList<E>` dispone de los siguientes métodos:

- Dos métodos interrogadores `size` que devuelve el tamaño y `isEmpty` que devuelve `true` si el tamaño de la lista es 0
- Un método de inserción `add` y otro de modificación `set` que cambia el elemento en una cierta posición
- Dos métodos de borrado `remove`, que borra un elemento, y `removeElementAt`, que borra el elemento que ocupa una cierta posición
- dos métodos de consulta, `get`, que accede por índice a un elemento y `indexOf`, que devuelve el índice de un cierto elemento

Los métodos que reciben un índice evalúan si el índice recibido se encuentra dentro del rango del array y en caso contrario lanzan la excepción `IndexOutOfBoundsException`.

■ **Ejemplo 2.1** El siguiente método código utiliza algunos métodos de la lista

```
IndexedList<Integer> list = new ArrayIndexedList<>();

list.add(0, 4);           // [4]
list.add(1, 5);           // [4, 5]
list.add(0, 1);           // [1, 4, 5]
list.add(1, 8);           // [1, 8, 4, 5]

show(list);

System.out.println("La posicion del 4 es: " + list.indexOf(4));

System.out.println("La posicion del 14 es: " + list.indexOf(14));

System.out.println("El que ocupa la posición 3 es el: " + list.get(3));

System.out.println("Ahora borramos");

list.remove(4);           // [1, 8, 5]
list.removeElementAt(0);  // [8, 5]

show(list);

System.out.println("Size: " + list.size());
System.out.println("IsEmpty: " + list.isEmpty());

}
```

Y utiliza el método `show`, que imprime todos los elementos de una lista y su correspondiente índice:

```
public static <E> void show(IndexedList<E> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println "[" + i + "]: " + list.get(i);
    }
}
```

Este otro método busca en una lista indexada si un elemento está en la lista. Obsérvese con cuidado el método `eqNull` por si la lista contiene elementos `null` y así evitamos la temida `NullPointerException`.

```
public static <E> boolean member (IndexedList<E> list, E elem) {
    int i = 0;
    while (i < list.size() && !eqNull(list.get(i), elem)) {
        i++;
    }
    return i < list.size();
}

public static boolean eqNull (Object o1, Object o2) {
    return o1 == o2 || o1 != null && o1.equals(o2);
}
```


3. Pilas (LIFO) y Colas (FIFO)

Las pilas o LIFOs (*stacks*) y las colas FIFO (*queues*) son TADs fundamentales que a pesar de su sencillez tienen innumerables aplicaciones, desde la propia ejecución de los programas hasta su uso en navegadores web, dispositivos móviles, sistemas operativos, hardware, etc.

3.1 Pila - LIFO (Last In First Out)

Una pila (*stack*) o LIFO es un TAD contenedor que consiste en una secuencia lineal de elementos de forma que el último elemento apilado (“push”) es el primero en ser desapilado (“pop”). Por ejemplos, pensad en una pila de platos, ¿cuál es el primer plato que vas a sacar?

Escribiremos las pilas horizontalmente de izquierda a derecha, siendo el primer elemento apilado el más a la izquierda. También se suelen escribir verticalmente, siendo el primer elemento apilado el más profundo: Por ejemplo en una pila apilamos A, después B y finalmente C:

```
+---+---+---+---+---+
| C | B | A |   |   | ...
+---+---+---+---+---+

      .
      .
      .
    |---|
    |   |
    |---|
    |   |
    |---|
    | C |
    |---|
    | B |
    |---|
    | A |
    |---|
```

No existe una operación de búsqueda. Sólo se puede apilar, desapilar, o devolver el elemento en la cima de la pila (`top`). Igual que en el caso de las listas, el tamaño de la pila está acotado por la memoria disponible del computador. Podrá producirse una excepción de insuficiente memoria al invocar el método `push`, lo que en el caso de la pila se conoce como “desbordamiento de pila” o “stack overflow”. (Ahora ya sabéis de dónde le viene el nombre a vuestra página web de consulta preferente, después de Google creo, en cuestiones de programación.)

Veremos dos formas de implementar pilas: usando listas de posiciones y usando arrays.

Interfaz **LIFO<E>**

El código del interfaz `'LIFO<E>'` es el siguiente:

```
public interface LIFO<E> extends Iterable<E> {

    public int size();

    public boolean isEmpty();

    public E top() throws EmptyStackException;

    public E pop();

    public void push(E elem);

    public Object [] toArray();

    public E [] toArray(E[] a);
}
```

Algunos comentarios sobre los métodos del interfaz:

- Interrogadores: `size`, `isEmpty`.
- Inserción: `push` apila un elemento.
- Borrado: `pop` desapila un elemento y lo devuelve.
- Acceso (“getters”): `top` devuelve el elemento en la cima de la pila. El comportamiento de `top` es equivalente a desapilar (`pop`) y volver a apilar (`push`) el elemento.

■ **Ejemplo 3.1** El siguiente método indica si en una cadena de caracteres que contiene una expresión de Java los paréntesis están equilibrados. El método recorre la cadena. Los paréntesis izquierdos se apilan. Al encontrar un paréntesis derecho la pila debe no ser vacía y se desapila el paréntesis izquierdo que le corresponde. Al terminar de recorrer la cadena la pila debe ser vacía.

Por ejemplo, dada la cadena `"((3 * 5) + (8 * (5 - 1)))"`:

	pila
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	(
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	((
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	(
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	((
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	(((
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	((
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	(
<code>((3 * 5) + (8 * (5 - 1)))</code> ^	

El código del método:

```
public static boolean balance(String str) {
    LIFO<Character> lifo = new .... ;
    boolean error = false;
```

```

    for (int i = 0; i < str.length() && !error; i++)
        switch (str.charAt(i)) {

            case '(': lifo.push('(');    // para equilibrar con ')'
                break;

            case ')': if (lifo.isEmpty()) error = true;
                    else lifo.pop();
                break;

            default: break;
        }
    return !error && lifo.isEmpty();
}

```

Observad que no importa el elemento que se apila. Lo que importa es si la pila está o no vacía al encontrar un paréntesis derecho y al terminar de recorrer la cadena. Se podría haber apilado un asterisco, un entero, etc.

3.2 Colas FIFO (First In First Out)

Una cola (“queue”) FIFO es un TAD contenedor que consiste en una secuencia lineal de elementos de forma que el primer elemento “encolado” (*enqueue*) es el primero en ser “desencolado” (*dequeue*), ..., y el último elemento encolado es el último en ser desencolado. (Sí, aquí estamos haciendo un uso anglicista y espurio de la palabras .^{en}colarz “desencolar”.) Pensad en la cola de la parada del bus. El primero en llegar es el primero en subir ... y el último en llegar es el último en subir.

Escribiremos las colas horizontalmente de izquierda a derecha, siendo el primer elemento encolado el más a la izquierda. Cola en la que se encola A, después B y finalmente C:

```

+---+---+---+---+---+
| A | B | C |   |   |
+---+---+---+---+---+

```

y el estado de la cola al desencolar:

```

+---+---+---+---+---+
| B | C |   |   |   |
+---+---+---+---+---+

```

Al igual que pasa con las pilas, No existe una operación de búsqueda. Sólo se puede encolar, desencolar, o devolver el elemento a desencolar (‘first’). Al igual que con las listas y las pilas, el tamaño de la cola está acotado por la memoria del computador. (Por razones estéticas no diremos que hay “desbordamiento de cola” cuando la cola esté llena.)

Veremos dos formas de implementar colas: usando listas de posiciones y usando arrays.

Interfaz FIFO<E>

El interfaz FIFO<E> es el siguiente:

```

public interface FIFO<E> extends Iterable<E> {

    public int size();

    public boolean isEmpty();

    public E first() throws EmptyFIFOException;

    public void enqueue(E elem);

    public E dequeue() throws EmptyFIFOException;

    public Object [] toArray();

    public E [] toArray(E[] a);
}

```

Algunos comentarios sobre el interfaz:

- El interfaz `FIFO<E>` extiende `Iterable<E>`:
- Interrogadores: `size`, `isEmpty`.
- Inserción: `enqueue` encola un elemento.
- Borrado: `dequeue` desencola un elemento.
- Acceso (“getters”): `first` devuelve el primer elemento a desencolar de la cola. El comportamiento de `first` NO es equivalente a desencolar y volver a encolar el elemento. Esto sólo se cumple cuando el elemento es el único de la cola.
- Conversión: el método `toArray` devuelve un vector (array) con los elementos de la cola. Devuelve un vector de `'Object'`. El método `toPositionList` devuelve una lista de posiciones con los elementos de la cola. Siguiendo la convención descrita anteriormente el primer elemento del vector y de la lista será el primer elemento desencolado, etc.

■ **Ejemplo 3.2** Supongamos que la variable `fifo` referencia una cola FIFO de objetos de clase `Process` de procesos del sistema operativo. La clase `Process` ofrece entre otros los métodos `run` y `suspend`. El método `run` permite ejecutar un proceso durante un número máximo de milisegundos indicados como parámetro. El proceso puede parar su ejecución antes de ese tiempo, bien porque ha terminado, bien porque ha recibido alguna interrupción o está a la espera de un evento, etc. El método `run` devuelve un número negativo si el proceso ha terminado. El método `suspend` salva el estado de un proceso parado que no ha terminado de forma que pueda reanudarse su ejecución en la próxima invocación a `run`. El siguiente bucle ejecuta los procesos con planificación circular o “round robin”.

```
while (!fifo.isEmpty()) {  
    Process p = fifo.first();  
    fifo.dequeue();  
    int status = p.run(20);  
    if (status < 0) {  
        p.suspend();  
        fifo.enqueue(p);  
    }  
}
```

Este ejemplo es meramente ilustrativo. En un sistema operativo real hay comunicación y sincronización entre procesos y la planificación es más sofisticada.

4. Complejidad

Nos referimos a la **complejidad** de un programa como una medida de su tiempo de ejecución o del uso de memoria. Sinónimos: eficiencia, coste. Podemos hablar de diferentes otros tipos de complejidad, pero en la asignatura hablaremos de dos tipos:

- Complejidad **temporal**: complejidad del tiempo de ejecución.
- Complejidad **espacial**: complejidad del espacio de memoria utilizado.

La complejidad en tiempo y la complejidad en espacio suelen estar reñidas: se ahorra tiempo a costa de usar más espacio, y se ahorra espacio a costa de tardar más tiempo. Es frecuente poder almacenar cálculos temporales para no volverlos a hacer, ganas velocidad, pero, a cambio, gastas más memoria.

La complejidad en tiempo o en espacio de un programa se calcula en *función del tamaño o de los valores de los datos de entrada* (abreviado “tamaño de la entrada”, “input size”). No se trata de todos los datos que toma o usa el programa. Se trata del tamaño de aquellos datos que son utilizados por el programa para resolver el problema y que tienen un impacto en el coste en tiempo o en espacio.

Al crecer el tamaño de la entrada la complejidad del programa o bien se mantiene constante o bien crece en función de dicho tamaño. No puede decrecer, incluso si se usa más espacio, porque se introduce el problema de buscar en el

Por n método que indica si un elemento está o no en un array. El “tamaño de la entrada” en este caso es el tamaño del array donde se hace la búsqueda. Cuanto más grande el tamaño del array más tarda el método en encontrar el elemento. El *tamaño del elemento* no es importante, incluso si se trata de un dato de gran tamaño en bits, pues sólo afecta la complejidad en tiempo total en un valor constante, es decir, sólo suma un valor constante al valor de complejidad total.

■ **Ejemplo 4.1** Un método que toma un elemento 'elem' y dos arrays 'arr1' y 'arr2' tal que 'arr2.length >= arr1.length'. El método busca 'elem' en 'arr1'.

- Si no está entonces termina y devuelve null.
- Si está, sea 'i' la posición de 'arr1' tal que 'elem == arr1[i]', entonces el método devuelve el elemento en 'arr2[i]'.

Aquí el “tamaño de la entrada” es 'arr1.length' pues el segundo array no es usado en la búsqueda. Cuanto más grande es el tamaño de 'arr1' más tarda el método en encontrar el elemento. El tamaño de 'arr2' no afecta el tiempo, pues el acceso directo a un array se asume que tiene complejidad en tiempo constante (es constante respecto al tamaño del array) ■

No sólo el tamaño de la entrada es relevante, también la *distribución de los datos*. Por ejemplo, el método que busca un elemento en un array no tarda lo mismo si cada vez pasamos arrays más grandes pero:

- El elemento a buscar es siempre el primero.
- El elemento a buscar está por el medio, o a veces está el primero y a veces en medio y a veces está el último o no está (y consideramos estos casos como equiprobables).
- El elemento a buscar está siempre el último o no está.
- El elemento a buscar está más veces el primero que en otra posición.

Podemos decir que se estudia la complejidad en *caso mejor, caso (pro)medio, caso peor, y caso amortizado*.

El estudio del *caso peor* nos da unos resultados más precisos y con menor variabilidad que los estudios de casos medios y experimentales. Los programas eficientes en el caso peor lo serán también en los otros casos. Para ello hay que considerar cuál es la entrada que daría el caso peor. En el ejemplo anterior, la peor entrada son arrays donde el elemento está siempre el último o no está, pues el programa siempre recorre el array completo.

Para hablar de complejidad habitualmente nos referimos a la **Complejidad asintótica**: idealmente el tamaño de la entrada podría ir creciendo indefinidamente (p.ej., se podrían ir pasando arrays más y más grandes a los métodos anteriores). La complejidad por tanto se calcula como el tiempo o espacio requerido en función del tamaño de la entrada cuando éste *tiende a infinito*. De esta forma, al hacerlo cuando tiene a infinito estamos ignorando los valores constantes o los valores nos significativos para la complejidad. Por ejemplo, si tenemos un método de complejidad $n^3 + n$, en el infinito podemos ignorar n y quedarnos sólo con n^3 . Para ello, la notación $O()$ nos permitirá dar una medida de complejidad.

Podemos decir que hay dos maneras de calcular la complejidad:

1. **Análisis experimental**: Se escribe el programa en un lenguaje de programación concreto. Requiere análisis estadístico (con buen muestreo de datos de entrada) y probabilístico (distribución de datos). Los resultados dependen del *entorno de ejecución* (lenguaje de programación, compilador, sistema operativo, arquitectura, entorno dinámico, etc.)
2. **Análisis teórico**: Se estudian los programas independientemente del entorno de ejecución. El lenguaje utilizado puede ser pseudocódigo, una notación que abstrae (alto nivel) las particularidades de un *paradigma* de programación.

Se asume que la complejidad de las operaciones del lenguaje pseudocódigo es proporcional en una constante a la complejidad real de su implementación en un lenguaje de programación concreto y ejecutándose en entorno concreto. Por tanto, la complejidad del programa en pseudocódigo es proporcional en una constante a la complejidad de su implementación. Dicho de otro modo, los factores a tener en cuenta en una implementación real afectan la complejidad teórica en un factor constante.

De esta forma, podemos estudiar programas en un entorno de ejecución concreto (p.ej., programa escrito en Java compilado con Eclipse y ejecutándose en las Aulas Informáticas) asumiendo que la complejidad del entorno es una función constante del programa en pseudocódigo.

En la práctica se necesita tanto el análisis teórico como el experimental: "In theory there is no difference between theory and practice. In practice there is." (Yogi Berra)

En lo tocante a TADs, las medidas de complejidad están asociadas a los métodos de las clases (implementaciones), no a los interfaces. Un interfaz describe simplemente la cabecera del método (el qué) y no la implementación (el cómo).

Pero un interfaz puede exigir que ciertos métodos se implementen con una complejidad determinada, forzando así una representación y familia de algoritmos para el TAD. Finalmente, la elección de métodos en los interfaces puede sugerir o descartar ciertas implementaciones.

Complejidad asintótica

Hablamos de complejidad asintótica cuando ignoramos los valores constantes de las expresiones. Algunas de las funciones matemáticas más habituales que representan la complejidad son las siguientes:

- Función **constante**: $f(n) = c$ con c constante, p.ej., $f(n) = 10$.
- Función **polinómica**: $f(n) = c_1 * n^{e_1} + \dots + c_m * n^{e_m}$ donde c_i (coeficientes) y e_i (exponentes) son enteros, p.ej.,
 $f(n) = 2 * n^2 + 7 * n - 20$.

El grado del polinomio se define como el exponente de mayor valor. Por ejemplo, el polinomio anterior tiene grado 2.

Entre las funciones polinomiales encontramos la lineal (grado 1), cuadrática (grado 2) y cúbica (grado 3).

- Función **logarítmica**, será casi siempre en base 2, por tanto en vez de $\log_2(n)$ escribiremos $\log(n)$.
- Función **exponencial**: $f(n) = c^n$ con c constante, p.ej., $f(n) = 2^n$.

Las funciones $\log(n)$ y 2^n son inversas: $\log_2(x) = y \iff 2^y = x$.

$$\begin{aligned} - \log_2(2^x) &= x \\ - 2^{\log_2(x)} &= x \end{aligned}$$

- Función **n-log(n)**: $f(n) = n * \log_2(n)$
 - Sumas aritméticas y geométricas.
 - Razones de crecimiento.
 - Funciones techo ('ceiling') y suelo ('floor').

Veamos algunas cifras comparativas:

Función	$n = 32$	$n = 64$	$n = 128$
$\log_2(n)$	5	6	7
n	32	64	128
$n * \log_2(n)$	160	384	896
n^2	1,024	4,096	16,384
n^3	32,768	262,144	2,097,152
2^n	4,294,967,296	18,446,744,073,709,551,616	(no cabe)

Para hacernos una idea del tamaño:

2^n para $n = 128$: 340,282,366,920,938,463,463,374,607,431,768,211,456

y una más:

n^n para $n = 32$: 1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,976

Podemos hablar de *problemas intratables*: polinomiales a partir de un 'n' medio, exponenciales a partir de un 'n' relativamente bajo, etc.

Notación $O()$

La intuición de la función $O()$ es establecer la complejidad de una función de 'n' indicando otra función proporcional que la acota asintóticamente (ignorando los valores constantes).

Definición 4.0.1 — Notación $O()$. Sean f y g dos funciones de naturales a reales. Se dice que $f(n)$ es del orden de $g(n)$, o $f(n)$ es $O(g(n))$, si existe una constante real $c > 0$ y una constante natural $n_0 \geq 1$ tal que $f(n) \leq c * g(n)$ para $n \geq n_0$

De forma intuitiva podemos decir que a partir de un valor n_0 de la entrada, la función $c * g(n)$ acota los valores de la función $f(n)$ cuando n tiende a infinito. Nótese que $O(0)$ no tiene sentido.

Una escala de complejidad de menor a mayor:

1. Constante: $O(1)$
2. Logarítmica: $O(\log(n))$
3. Lineal: $O(n)$
4. N-Log-N: $O(n * \log(n))$
5. Cuadrática $O(n^2)$
6. Cúbica: $O(n^3)$
7. Polinomial: $O(n^m)$ de orden m
8. Exponencial: $O(2^n) \dots O(m^n)$

Algunos comentarios sobre las funciones:

- Todas las funciones constantes son $O(1)$: Sea $f(n) = c$ función constante. Entonces $f(n) \leq c * g(n)$ con $g(n) = 1$ y $n_0 = 1$.
- Todas las funciones polimórficas son $O(n^{e_1})$ donde e_1 es el exponente de mayor grado: Sea $f(n) = c_1 * n^{e_1} + \dots + c_m * n^{e_m}$ tal que e_1 es el exponente de mayor grado. Entonces existe un coeficiente c y un n_0 tal que $f(n) \leq c * n^{e_1}$ para $n \geq n_0$. Es decir, multiplicando n^{e_1} un número c de veces podemos hacer el resultado mayor que lo que vale el resto del polinomio $c_2 * n^{e_2} + \dots + c_m * n^{e_m}$, pues n^{e_1} crece mucho más rápido que n^{e_2}, \dots, n^{e_m} .
- En las funciones polinomiales, **se ignoran factores constantes y de orden inferior**. Esto viene justificado por el álgebra de $O()$:
 - $O(c * f(n))$ es $O(f(n))$
 - $O(f(n) + g(n))$ es $O(f(n))$ si $g(n)$ es $O(f(n))$
 - $O(f(n) + g(n))$ es $O(g(n))$ si $f(n)$ es $O(g(n))$

■ Ejemplo 4.2 Veamos algunos ejemplos de métodos con diferentes complejidades:

```
public static long contador = 0;

public static void logn (int n) {
    for (int i = n; i > 0; i = i/2) {
        contador ++;
    }
}
```

```
    }  
}  
  
public static void lineal (int n) {  
    for (int i = 0; i < n; i++) {  
        contador++;  
    }  
}  
  
public static void nlogn (int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = i; j > 0; j = j / 2) {  
            contador++;  
        }  
    }  
}  
  
public static void cuadratica (int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            contador++;  
        }  
    }  
}  
  
public static void cubica (int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            for (int k = 0; k < n; k++) {  
                contador++;  
            }  
        }  
    }  
}  
  
public static void polinmica4 (int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            for (int k = 0; k < n; k++) {  
                for (int l = 0; l < n; l++) {  
                    contador++;  
                }  
            }  
        }  
    }  
}  
  
public static void exponencial (int n) {  
    contador++;  
    if (n < 0) return ;  
    exponencial(n-2);  
    exponencial(n-1);  
}  
  
public static void exponencialN (int n) {  
    contador++;  
    if (n < 0) return;  
    for(int i = 0; i < n; i++) {  
        exponencialN(n-1);  
    }  
}
```


5. Ordenación

5.1 Órdenes Totales

En este tema estudiaremos el concepto matemático de **orden total**, también llamado **orden lineal**.

Vamos a empezar con una perspectiva teórica. Desde la teoría de conjuntos, una entrada es un par perteneciente al producto cartesiano de un conjunto K de claves y un conjunto V de valores: (k, v) está en $K \times V$, donde:

$$K \times V = \{(k, v) \mid k \in K \wedge v \in V\}$$

Relación de **orden total computable** \leq sobre el conjunto de claves K .

1. Hay una relación de igualdad computable (existe un programa que la computa) en K que permite determinar si una clave es igual o no a otra. (El requisito de computabilidad es obvio pero importante, pues según el famoso resultado de Alan Turing, existen funciones de naturales a naturales que no son computables, es decir, no existe un programa que compute sus valores.)

La relación de igualdad es una relación de equivalencia, es decir, cumple las siguientes reglas:

- Reflexiva: para toda clave k se cumple $k = k$.
- Simétrica: para toda k_1 y k_2 , si $k_1 = k_2$ entonces $k_2 = k_1$.
- Transitiva: para toda k_1, k_2 y k_3 , si $k_1 = k_2$ y $k_2 = k_3$ entonces $k_1 = k_3$.

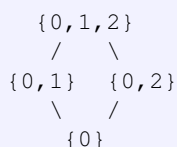
2. Hay una relación total computable (hay un programa que la computa) que satisface las siguientes reglas:

- Totalidad: para toda k_1 y k_2 , o bien $k_1 \leq k_2$ o bien $k_2 \leq k_1$, una de ellas debe ser cierta.
- Antisimétrica: para toda k_1 y k_2 , si $k_1 \leq k_2$ y $k_2 \leq k_1$ entonces $k_1 = k_2$.
- Transitiva: para toda k_1, k_2 y k_3 , si $k_1 \leq k_2$ y $k_2 \leq k_3$ entonces $k_1 \leq k_3$.

Totalidad y antisimétrica implican reflexiva.

Una relación computable entre dos elementos tiene asociado un programa que toma los dos elementos y devuelve un booleano que indica si el par está o no en la relación. El programa computa una relación de orden total si cumple las propiedades matemáticas de una relación de orden total. Algunos ejemplos de órdenes totales pueden ser: los números naturales con \leq o las cadenas de caracteres con orden lexicográfico.

■ **Ejemplo 5.1** Ejemplo de orden **parcial**: dados dos conjuntos S_1 y S_2 , definimos la relación \leq de la siguiente manera: $S_1 \leq S_2$ si y sólo si para todo x en S_1 se tiene que x está en S_2 . Dicha relación es una relación de orden pues cumple “reflexiva”, “antisimétrica” y “transitiva”, pero no cumple “totalidad” y por lo tanto es un orden parcial:



Por ejemplo con esto, sabemos:

- $\{0\} \leq \{0, 2\}$ es cierto y que $\{0, 2\} \leq \{0\}$ es falso.
- Pero $\{0, 1\} \leq \{0, 2\}$ es falso y $\{0, 2\} \leq \{0, 1\}$ es falso.

Un orden parcial define un retículo. Un orden total define una recta.

Todo orden total lleva asociado un *orden total estricto* $<$ que se define como $k_1 < k_2$ si y sólo si $k_1 \leq k_2$ y no $k_1 = k_2$. Se puede definir un orden estricto sobre un orden total porque tenemos igualdad.

En la práctica se desea tener conjuntos K acotados en los que hay una **clave mínima** y o bien una **clave máxima** o bien un **supremo**:

- Clave mínima: si para toda k se tiene $k_{min} \leq k$ entonces k_{min} es una clave mínima.
- Clave máxima: si para toda k se tiene $k \leq k_{max}$ entonces k_{max} es una clave máxima.
- Las claves mínimas y máximas son únicas porque el orden es total.
- Supremo: se toma la unión de K con un conjunto formado por un único elemento sup , se extiende la relación de orden a esa unión, y se postula que $k \leq sup$ para toda k .

5.2 Interfaces Comparable y Comparator

Ambos interfaces se utilizan para definir órdenes totales en Java. La diferencia está en cómo y para qué se usan.

5.2.1 Interfaz java.lang.Comparable

```
public interface Comparable<T> {
    public int compareTo(T t);
}
```

El interfaz `Comparable<T>` establece un orden *natural* entre los objetos de una clase. La implementación de `Comparable<T>` por la clase `'T'` hace que ésta tenga que implementar el método `compareTo` para comparar dos objetos de `'T'`. Ese método de comparación *queda fijo para todos los objetos de 'T' y se le llama orden natural*.

Dados `'t1'` y `'t2'` objetos de `'T'`, la invocación `t1.compareTo(t2)` debe devolver un entero `'i'` tal que:

```
i < 0    si t1 es menor que t2.
i == 0   si t1 es igual que t2.
i > 0    si t2 es menor que t1.
```

El comparador debe ser **consistente con equals** cuando se cumple que `t1.equals(t2)`, entonces `t1.compareTo(t2) == 0`. De la misma forma, si `t1.equals(t2)` también se debe cumplir la expresión `t1.compareTo(t2) == t2.compareTo(t1)`.

- **Ejemplo 5.2** La clase `String` de Java implementa `Comparable<String>` y por tanto implementa el método `'compareTo'` que se puede usar para comparar cadenas de caracteres. El método de comparación elegido computa el orden lexicográfico de las cadenas.

```
public class String implements Comparable<String> {
    public int compareTo(String s) {
        /* Comparación lexicográfica entre la cadena de este objeto y la
        * cadena 's' pasada como argumento
        */
        ... // código aquí que no nos interesa ahora
    }
}
```

- **Ejemplo 5.3** Ahora un ejemplo de uso:

```
public static void main(String [] s) {
    String s1 = "Terabyte";    /* 10^12 bytes. */
    String s2 = "Terapeuta";   /* 10^12 peutas. */

    int r = s1.compareTo(s2);
}
```

```

if (r < 0)
    System.out.println(s1 + " es menor que " + s2);
else if (r == 0)
    System.out.println(s1 + " es igual que " + s2);
else
    System.out.println(s1 + " es mayor que " + s2);
}

```

■ **Ejemplo 5.4** El siguiente método indica si una clave está en una lista de entradas, el código asume que Comparable<K> está implementado por la clase 'K' de las claves:

```

public static <K,V> boolean search(K k, Entry<K,V> [] array) {
    boolean found = false;
    int i = 0;
    while (i < array.length && !found) {
        found = array[i].getKey().compareTo(k) == 0 ;
        i++;
    }
    return found;
}

```

5.2.2 Interfaz java.util.Comparator

El interfaz Comparator<T> permite implementar objetos que son “operadores” de comparación. El interfaz Comparator<T> se define:

```

public interface Comparator<T> {
    public int compare(T t1, T t2);
}

```

Los objetos de aquellas clases que implementen el interfaz Comparator<T> implementan un método de comparación compare para objetos de la clase 'T'. Dados 't1' y 't2' objetos de 'T' y 'c' un objeto comparador, la invocación c.compare(t1,t2) debe devolver un entero 'i' tal que:

$i < 0$	si t1 es menor que t2.
$i == 0$	si t1 es igual que t2.
$i > 0$	si t2 es menor que t1.

El comparador debe ser **consistente con equals** cuando se cumple que t1.equals(t2), entonces c.compare(t1,t2) == 0. De la misma forma, si t1.equals(t2) también se debe cumplir la expresión c.compare(t1,t2) == c.compare(t2,t1).

Se pueden usar diferentes objetos comparadores para comparar objetos de clase 'T'. Por ejemplo: podemos definir varios comparadores para la clase String: comparador lexicográfico, tamaño, coste de compresión bajo zip, etc.

■ **Ejemplo 5.5** Una clase de comparadores de cadenas de caracteres que las comparan según su longitud:

```

public class SizeStringComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}

```

- **Ejemplo 5.6** Una clase de comparadores de cadenas de caracteres que las comparan según el orden lexicográfico:

```
public class LexicographicStringComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        int minLength = Math.min(s1.length(), s2.length());
        for (int i = 0; i < minLength && s1.charAt(i) == s2.charAt(i); i++) ;
        if (i < minLength) {
            return s1.charAt(i) < s2.charAt(i) ? -1 : 1 ;
        }
        else {
            return s1.length() - s2.length();
        }
    }
}
```

- **Ejemplo 5.7** Un ejemplo sencillo que usa ambos comparadores:

```
public static void main(String [] s) {
    String s1 = "Terabyte";
    String s2 = "Terapeuta";
    SizeStringComparator sc = new SizeStringComparator();
    LexicographicStringComparator lc = new LexicographicStringComparator();
    int r;

    r = lc.compare(s1,s2);
    if (r < 0)
        System.out.println(s1 + " es menor que " + s2);
    else if (r == 0)
        System.out.println(s1 + " es igual que " + s2);
    else
        System.out.println(s1 + " es mayor que " + s2);

    r = sc.compare(s1,s2);
    if (r < 0)
        System.out.println(s1 + " es menor que " + s2);
    else if (r == 0)
        System.out.println(s1 + " es igual que " + s2);
    else
        System.out.println(s1 + " es mayor que " + s2);
}
```

- **Ejemplo 5.8** El siguiente método indica si una clave está en una lista de entradas, el código recibe un comparador para la clase 'K' de las claves:

```
public static <K,V> boolean search(K k, Entry<K,V> [] array, Comparator<K> c){
    boolean found = false;
    int i = 0;
    while (i < array.length && !found) {
        found = c.compare(array[i].getKey(),k) == 0 ;
        i++;
    }
    return found;
}
```

■ **Ejemplo 5.9** Un comparador lexicográfico de puntos en dos dimensiones:

```
public class Lexicographic2DPointComparator implements Comparator<2DPoint> {
    private int xa, ya, xb, yb;
    public int compare(2DPoint a, 2DPoint b) {
        return a.getX() != b.getX() ? b.getX() - a.getX() : b.getY() - a.getY();
    }
}
```

Comparator<T> a partir de Comparable<T>

Si un objeto es de una clase que implementa el interfaz `Comparable<T>` entonces se puede crear un objeto `Comparator<T>` por defecto para dicha clase:

```
import java.util.Comparator;
import java.lang.Comparable;

public class DefaultComparator<T> implements Comparator<T> {
    public int compare(T a, T b) throws UnsupportedOperationException {
        if (a instanceof Comparable<?>) {
            return ((Comparable<T>) a).compareTo(b);
        }
        else {
            throw new UnsupportedOperationException();
        }
    }
}
```

Si 'T' implementa `Comparable<T>` entonces dispone de 'compareTo' que puede usarse para comparar. En otro caso se lanza la excepción.

```
public static void main (String [] arg) {
    DefaultComparator<String> c = new DefaultComparator<String>();
    String s1 = new String("aa");
    String s2 = new String("bb");
    System.out.println(c.compare(s1,s2));

    DefaultComparator<Object> o = new DefaultComparator<Object>();
    try {
        System.out.println(o.compare(new Object(), new Object()));
    } catch (UnsupportedOperationException e) {
        System.out.println("Object no implementa Comparable<Object>");
    }
}
```

El problema de este comparador por defecto es que la excepción se lanza al invocarse 'compare'. Sería mejor que la excepción se lanzase al crear el objeto comparador. Para hacer bien esto hay que usar trucos de Java, en particular el API de reflexión que nos permite pasar a un constructor un valor en tiempo de ejecución que representa el tipo 'T' de un elemento. El constructor puede entonces comprobar si ese tipo implementa `Comparable<?>` y lanzar la excepción en caso negativo.

6. Listas de Posiciones

Ya hemos visto un tipo de lista, la lista indexada `IndexedList<E>` en el Tema 2, en la que, para acceder a los elementos, se utiliza un índice numérico. En la asignatura veremos otro posible TAD de tipo lista: las **listas de posiciones**, y veremos los detalles de una implementación mediante cadenas de objetos nodo doblemente enlazados (“doubly-linked”), es decir, cada nodo que compone la lista tiene un puntero al siguiente elemento y puntero al elemento anterior. Las listas de posiciones no utilizan un índice numérico para el acceso a los elementos sino que utilizan como medio de acceso el nodo en el que están contenidos los elementos de la lista. La lista se puede recorrer solicitando el siguiente/anterior nodo de la lista hasta que ya no haya nodos pendientes de recorrer porque estemos en la última/primer posición.

¿Qué operaciones definen una lista?

De forma similar a lo descrito para los números complejos en el Tema 1, el TAD de listas vendrá dado por uno (o más) interfaces que definen la funcionalidad de la lista, y por una (o más) clases que implementan el interfaz. Primero nos centramos en el interfaz y en el uso del mismo, ilustrando así la importancia y los beneficios de la abstracción de datos: escribiremos código que únicamente *usa* el interfaz de listas de posiciones sin saber realmente cómo se implementan dichas listas. Podemos decir que este código sería un “cliente” del TAD lista:

De acuerdo a lo visto en Programación II, observamos que la inserción, búsqueda y borrado de elementos usando índices puede ser un poco diferente al uso de un array:

```
list.insert(0, "vais");           // ["vais"]

list.insert(1, "a");              // ["vais", "a"]

list.insert(2, "aprobar");        // ["vais", "a", "aprobar"]

list.insert(0, "no");             // ["no", "vais", "a", "aprobar"]

list.set(3, "suspender");         // ["no", "vais", "a", "suspender"]

list.remove(2);                  // ["no", "vais", "suspender"]

list.set(1, "quiero");           // ["no", "quiero", "suspender"]
```

Los métodos ‘insert’, ‘remove’ y ‘set’ tienen que posicionarse en el nodo con número correspondiente, y el programador debe recordar y llevar actualizada la correspondencia entre números y nodos. Además está el tema del control de rango, p.ej., si hacemos `list.set(200, "Fuera de rango")` tenemos un problema.

Nuestro objetivo es poder insertar y borrar los nodos directamente sin comprometer la abstracción de datos, es decir, sin que el cliente de la lista tenga que conocer cómo se implementa un nodo ni cómo se enlaza en la lista.

6.1 Los interfaces: Position y PositionList

Interfaz Position<E>

Una `Position` es una abstracción de un nodo, es decir, el interfaz `Position<E>` ofrece un único método `element` que debe devolver el elemento almacenado en el nodo. Cualquier otra cosa no le interesa al cliente de la lista. Otra cosa es la implementación del interfaz, que veremos más adelante.

```
public interface Position<E> {
    public E element ();
}
```

Este interfaz se utilizará tanto para listas de posiciones como para otros TADs que veremos más adelante.

Interfaz PositionList<E>

El interfaz `PositionList<E>` utiliza el interfaz `Position<E>` para abstraer la clase de los nodos: los objetos nodo de la lista serán objetos de una clase que implemente `Position<E>`. De una posición únicamente podemos obtener el elemento que almacena. Recordad que `Position` (posición) significa “nodo abstracto”.

Nótese que el interfaz `PositionList<E>` extiende `Iterable<E>`: de momento ignoramos ésto, ya volveremos sobre ello en el tema de Iteradores.

```
public interface PositionList<E> extends Iterable<E> {

    public int size();

    public boolean isEmpty();

    public Position<E> first();

    public Position<E> last();

    public Position<E> next(Position<E> p) throws IllegalArgumentException;

    public Position<E> prev(Position<E> p) throws IllegalArgumentException;

    public void addFirst(E elem);

    public void addLast(E elem);

    public void addBefore(Position<E> p, E elem) throws IllegalArgumentException;

    public void addAfter(Position<E> p, E elem) throws IllegalArgumentException;

    public E remove(Position<E> p) throws IllegalArgumentException;

    public E set(Position<E> p, E elem) throws IllegalArgumentException;

    public Object [] toArray();

    public E [] toArray(E[] a);
}
```

La documentación completa sobre los métodos la podéis encontrar en el javadoc de la clase `PositionList`. A modo de resumen, algunos comentarios sobre los métodos:

- Interrogadores: `size`, `isEmpty`.
- Acceso (“getters”): `first`, `last`, `next` y `prev`. Permiten moverse por las posiciones de la lista. Devuelven ‘null’ si no existe el nodo al que se pretende acceder. De esta forma se puede usar null como valor centinela en bucles.
- Inserción: `addFirst`, `addLast`, `addBefore` y `addAfter`.
- Borrado: `remove` borra una posición, dejando las otras posiciones “conectadas”.
- Modificación: `set` modifica el elemento de una posición.
- Conversión: el método `toArray` devuelve un vector (array) con los elementos de la lista en el mismo orden posicional de izquierda a derecha. Devuelve un vector de ‘Object’ (mucho cuidado con los arrays de genéricos, para tratarlos Java tiene “serios” problemas).
- La excepción ‘`IllegalArgumentException`’ será lanzada cuando la posición pasada al método sea null o no sea una posición de la lista (p.ej. cuando sea un nodo de otra lista).

Una forma de entender el uso del interfaz es ver una lista como un objeto al que le podemos pedir cajas (con `first` o `last` de las cuales sólo podemos sacar el elemento que contienen (método `element` del interfaz `Position<E>`). Tenemos que darle al objeto lista la caja que tenemos para que nos de la siguiente o la anterior. También tenemos que darle la caja para cambiar el elemento. El interfaz no especifica si la inserción conlleva *copia superficial* ("shallow copy") o *copia profunda* ("deep copy") de los elementos. Con la copia superficial se insertan (las referencias a) los objetos elemento en la lista, mientras que con la copia profunda se insertan (las referencias a) copias de los objetos elemento en la lista. Como norma general las estructuras de datos utilizan la copia superficial. Todos los interfaces que veremos en la asignatura asumen copia superficial, dejando al usuario del interfaz la responsabilidad de hacer copias profundas. Para poder hacer copia profunda los objetos elemento deben tener constructor de copia (o el método 'clone', si bien el uso de dicho método es cuestionado, ved por ejemplo:

```
http://www.javapractices.com/topic/TopicAction.do?Id=71).
```

El uso de copia superficial puede ser problemático ya que la referencia al objeto elemento podría estar compartida. A este problema se le llama *aliasing*.

Por ejemplo, supongamos que insertamos un mismo contacto en dos listas distintas.

```
Contacto c = new Contacto("LuisMi");

list1.addFirst(c);
list2.addFirst(c);

c.setName("LuisMa");
```

Ambas listas comparten el mismo elemento que ha quedado modificado desde "fuera" de la lista.

Además, el elemento puede modificarse desde "dentro" de la primera lista quedando también modificado en la segunda:

```
Contacto c = new Contacto("LuisMi");

list1.addFirst(c);
list2.addFirst(c);

list1.first().element().setName("Luisito");
```

Para evitar alias el usuario del interfaz debe hacer copia profunda explícita con el constructor de copia:

```
Contacto c = new Contacto("LuisMi");

list1.addFirst(new Contacto(c));
list2.addFirst(new Contacto(c));

c.setName("LuisMa");
list1.first().element().setName("Luisito")
```

Ahora 'c' referencia un objeto "LuisMa", el primer elemento de 'list1' referencia un objeto "Luisito" el primer elemento de 'list2' referencia un objeto "LuisMi".

Ejemplo de uso

Ahora actuaremos como clientes que quieren usar el TAD de listas, es decir, únicamente disponemos del interfaz `Position<E>` y de `PositionList<E>` que documenta lo que hace cada método. Asumimos que disponemos de una clase que implementa el interfaz. De momento desconocemos los detalles de dicha clase (los veremos más adelante), y como clientes del interfaz, realmente no nos interesan. Nos basta con conocer los constructores. Se nos dice que hay un constructor que crea una lista vacía (sin nodos). Podemos empezar a escribir código sin saber nada acerca de los nodos o la implementación, salvo que tenemos a nuestra disposición una clase `NodePositionList<E>` con un constructor que construye una lista vacía.

```
PositionList<String> list = new NodePositionList(); // lista vacia

Position<String> cursor; // cursor a nodos

list.addFirst("vais"); // ["vais"]

list.addLast("a"); // ["vais", "a"]
```

```

list.addLast("aprobar");           // ["vais", "a", "aprobar"]

list.addFirst("no");               // ["no", "vais", "a", "aprobar"]

cursor = list.last();

list.set(cursor, "suspender");     // ["no", "vais", "a", "suspender"]

cursor = list.prev(cursor);

list.remove(cursor);               // ["no", "vais", "suspender"]

cursor = list.prev(list.last());

list.set(cursor, "quiero");        // ["no", "quiero", "suspender"]

```

El programador tiene que seguir llevando cuenta de dónde está cada elemento, pero no es necesario recordar la correspondencia entre nodos e índices. Además, los métodos no tienen que empezar a operar desde el comienzo o el fin de la lista sino desde la posición dada.

Recorridos de listas de posiciones

- **Ejemplo 6.1** Mostrar por pantalla todos los elementos de una lista. Observad el uso de **null** como posición *centinela* en la condición del bucle. Observad que usamos *métodos genéricos* en los ejemplos.

```

// con bucle while

public static <E> void show(PositionList<E> list) {
    Position<E> cursor = list.first();
    while (cursor != null) {
        System.out.println(cursor.element());
        cursor = list.next(cursor);
    }
}

// con bucle for
public static <E> void show(PositionList<E> list) {
    for (Position<E> cursor = list.first();
        cursor != null;
        cursor = list.next(cursor))
        System.out.println(cursor.element());
}

```

Nota: éstos métodos son estáticos y genéricos porque no están dentro de la clase que implementa el interfaz.

Consideraciones generales

- Se recorren las listas de forma secuencial usando bucles y nodos cursor.
- La inicialización consistirá en hacer que el cursor referencie la primera (o la última) posición de la lista. Si la lista está vacía entonces el cursor será **null**.
- La condición de parada (la contraria de la de continuación) del bucle la marca el problema, pero suele incluir la condición de control de rango: el cursor debe ser distinto de **null**.
- El incremento consistirá en avanzar el cursor a la siguiente (o anterior) posición.
- Frecuentemente se usan iteradores para recorrer listas, pero para entender cómo funcionan tenemos que aprender a recorrer listas mediante cursores. Un iterador es una abstracción de un cursor.
- Cuidado: en ocasiones `cursor.element().equals(x)` puede ser preferible a `x.equals(cursor.element())`. Hay que tener en cuenta si la lista puede contener elementos **null**, y si los elementos a comparar (los 'x') pueden ser **null**. Si la lista no contiene elementos **null** entonces '`cursor.element().equals(x)`' es seguro. En cualquier caso esto sería seguro:

```

cursor.element() != null && cursor.element().equals(x)

```

■ **Ejemplo 6.2** Insertar un elemento antes de la primera aparición de otro. El siguiente método toma como argumento una lista 'list' y dos elementos 'a' y 'e'. El método añade a la lista el elemento 'a' antes de la primera aparición en la lista del elemento 'e'. Si 'e' no está en la lista entonces el método deja la lista intacta:

```
public static <E> void addBeforeElement(PositionList<E> list, E e, E a) {
    Position<E> cursor;
    for (cursor = list.first();
        cursor != null && !cursor.element().equals(e);
        cursor = list.next(cursor))
        ; // No hay cuerpo del bucle

    // cursor == null || cursor != null && cursor.element().equals(e);
    if (cursor != null) list.addBefore(cursor, a);
}
}
```

■ **Ejemplo 6.3** Borrar todas las apariciones de un elemento dado. Recordad algo importante: *antes de borrar hay que avanzar*.

```
public static <E> void deleteAll(E elem, PositionList<E> list) {
    Position<E> cursor = list.first();
    while (cursor != null) {
        if (cursor.element().equals(elem)) {
            Position<E> borrar = cursor;
            cursor = list.next(cursor); // avanza
            list.remove(borrar);       // borra
        } else {
            cursor = list.next(cursor); // avanza
        }
    }
}
```

Variante que invoca list.next() una vez:

```
public static <E> void deleteAll(E elem, PositionList<E> list) {
    Position<E> cursor = list.first();
    while (cursor != null) {
        Position<E> nxt = list.next(cursor); // avanza
        if (cursor.element().equals(elem))
            list.remove(cursor); // borra
        cursor = nxt;
    }
}
```

Algunas variantes del condicional anterior que NO FUNCIONAN:

```
if (cursor.element().equals(elem)) {
    list.remove(cursor);
    cursor = list.next(cursor); // IllegalArgumentException
} ...
```

```
if (cursor.element().equals(elem)) {
    cursor = list.next(cursor);
    list.remove(list.prev(cursor)); // IllegalArgumentException
} ...
```

```
if (cursor.element().equals(elem)) {
    Position<E> borrar = cursor;
    list.remove(cursor);
}
```

```

        cursor = list.next(borrar);           // IllegalArgumentException
    } ...

```

■ **Ejemplo 6.4** El peligro de usar contadores de elementos: usar un contador de elementos recorridos no es necesario y puede ser peligroso dependiendo del problema a resolver:

```

// no funciona
public static <E> void show(PositionList<E> list) {
    for (int i = 0; i < list.size(); i++)
        System.out.println(cursor.element());
}

// redundante
public static <E> void show(PositionList<E> list) {
    Position<E> cursor = list.first();
    for (int i = 0; i < list.size(); i++) {
        System.out.println(cursor.element());
        cursor = list.next(cursor);
    }
}

// redundante y no funciona
public static <E> void deleteAll(E elem, PositionList<E> list) {
    Position<E> cursor = list.first();
    int i = 0;
    while (i < list.size()) {
        Position<E> nxt = list.next(cursor);
        if (cursor.element().equals(elem))
            list.remove(cursor);
        cursor = nxt;
        i++;
    }
}

```

■ **Ejemplo 6.5** Intercalar los elementos de dos listas hasta que acabe la más pequeña. El siguiente método toma como parámetros dos listas y devuelve una nueva lista con los elementos de las dos listas intercalados hasta terminar de recorrer la lista más pequeña. Si alguno de los parámetros es null o referencia una lista vacía entonces el método devuelve una nueva lista vacía.

```

public static <E> PositionList<E> shuffleShorter(PositionList<E> list1,
                                                  PositionList<E> list2) {

    PositionList<E> result = new NodePositionList<E>();

    if (list1 == null || list2 == null || list1.isEmpty() || list2.isEmpty())
        return result;

    Position<E> cursor1 = list1.first();
    Position<E> cursor2 = list2.first();

    while (cursor1 != null && cursor2 != null) {
        result.addLast(cursor1.element());
        result.addLast(cursor2.element());
        cursor1 = list1.next(cursor1);
        cursor2 = list2.next(cursor2);
    }
}

```

```

    return result;
}

```

Ejercicio 6.1 Modificar el código para que añada al final de la lista resultado los elementos sobrantes de la lista de mayor tamaño. Hay que modificar también la condición frontera: si una lista es null o vacía y la otra no entonces el método debe devolver una copia de la lista no null y no vacía.

■ **Ejemplo 6.6** Invertir los elementos de una lista. El siguiente método toma una lista y devuelve una nueva lista con los elementos de la primera en orden inverso. Si la lista parámetro es null entonces el método devuelve null. Si la lista parámetro es vacía entonces el método devuelve una nueva lista vacía.

```

public static <E> PositionList<E> reverseList(PositionList<E> list) {
    if (list == null) return null;
    PositionList<E> result = new NodePositionList<E>();
    if (list.isEmpty()) return result;

    Position<E> cursor = list.first();
    while (cursor != null) {
        result.addFirst(cursor.element());
        cursor = list.next(cursor);
    }

    /* Otra posibilidad es recorrer la lista de derecha a izquierda:

    Position<E> cursor = list.last();
    while (cursor != null) {
        result.addLast(cursor.element());
        cursor = list.prev(cursor);
    }

    */

    /* Podemos en ambos casos usar un bucle for, por ejemplo:

    for (Position<E> cursor = list.last();
        cursor != null;
        cursor = list.prev(cursor))
        result.addLast(cursor.element());

    */

    return result;
}

```

■ **Ejemplo 6.7** El siguiente método invierte los elementos de la lista parámetro directamente sin devolver una nueva lista. Usa los métodos remove y addFirst y por tanto borra nodos (no todos) de la lista y crea nuevos nodos. Recorre la lista desde el final.

```

public static <E> void reverseInPlace(PositionList<E> list) {
    if (list != null && !list.isEmpty() && list.size() > 1) {
        Position<E> cursor = list.last();
        Position<E> end = list.first();
        while (cursor != end) {
            Position<E> prv = list.prev(cursor);
            E element = list.remove(cursor);

```

```

        list.addFirst(element);
        cursor = prv;
    }
}

```

La condición frontera `list.size() > 1` se añade por claridad, pero el código funciona igualmente sin ella.

Si modificamos el código anterior para recorrer la lista desde el principio el resultado es erróneo, tanto si se usa `addFirst` como `addLast`.

```

public static <E> void reverseInPlace(PositionList<E> list) {
    if (list != null && !list.isEmpty() && list.size() > 1) {
        Position<E> cursor = list.first(); // MAL
        Position<E> end = list.last();
        while (cursor != end) {
            Position<E> nxt = list.next(cursor); // MAL
            E element = list.remove(cursor);
            list.addLast(element); // MAL
            cursor = nxt;
        }
    }
}

```

Una versión correcta que recorre la lista desde el principio. Usa 'addAfter' sobre el último nodo.

```

public static <E> void reverseInPlace(PositionList<E> list) {
    if (list != null && list.isEmpty() && list.size() > 1) {
        Position<E> cursor = list.first();
        Position<E> end = list.last();
        while (cursor != end) {
            Position<E> nxt = list.next(cursor);
            E element = list.remove(cursor);
            list.addAfter(end, element);
            cursor = nxt;
        }
    }
}

```

Una versión alternativa que surge de observar en la solución anterior que el nodo a borrar (el que referencia `cursor`) siempre es el primero, y que después del borrado la variable `cursor` referencia el objeto que referencia 'nxt', que siempre es el primero. Por tanto `cursor` y `nxt` no son necesarios y se puede usar directamente `list.first()`.

```

public static <E> void reverseInPlace(PositionList<E> list) {
    if (list != null && list.isEmpty() && list.size() > 1) {
        Position<E> end = list.last();
        while (list.first() != end) {
            list.addAfter(end, list.first().element());
            list.remove(list.first());
        }
    }
}

```

Esta versión invierte los elementos de la lista sin crear nodos nuevos. Modifica los elementos de los nodos existentes.

```

public static <E> void reverseSwap(PositionList<E> list) {
    if (list != null && list.isEmpty() && list.size() > 1) {
        Position<E> start = list.first();
        Position<E> end = list.last();
    }
}

```

```

        while (start != end) {
            // swap elements
            E tmp = start.element();
            list.set(start,end.element());
            list.set(end,tmp);
            // move
            if (list.next(start) == end)
                start = end;
            else {
                start = list.next(start);
                end = list.prev(end);
            }
        }
    }
}

```

■ **Ejemplo 6.8** Borrar un cierto rango de valores de una lista. El siguiente método borra de una lista desde el nodo número 'i' hasta el nodo número 'j'. Los nodos se numeran de 0 a list.size()-1.

```

public static <E> void removeRange(PositionList<E> list,
                                   int i,
                                   int j)
    throws IllegalArgumentException {

    if (! (0 <= i && i <= j && j < list.size()) )
        throw new IllegalArgumentException("Invalid range");

    if (list != null && !list.isEmpty()) {
        Position<E> cursor = list.first();
        int pos = 0;
        while (pos != i) { // posicionarse
            cursor = list.next(cursor);
            pos++;
        }
        while (pos <= j) { // borrar
            Position<E> nxt = list.next(cursor);
            list.remove(cursor);
            cursor = nxt;
            pos++;
        }
    }
}

```

La siguiente variación se posiciona y borra en un único bucle:

```

public static <E> void removeRange(PositionList<E> list,
                                   int i,
                                   int j)
    throws IllegalArgumentException {

    if (! (0 <= i && i <= j && j < list.size()) )
        throw new IllegalArgumentException("Invalid range");

    if (list != null && !list.isEmpty()) {
        Position<E> cursor = list.first();
        int pos = 0;
        while (pos <= j) {
            Position<E> nxt = list.next(cursor);
            if (pos >= i) list.remove(cursor);
            cursor = nxt;
            pos++;
        }
    }
}

```

```

    }
  }
}

```

■ **Ejemplo 6.9** Cortar la lista a un cierto tamaño. El siguiente método modifica una lista para quedarse con los 'size' primeros nodos. Si 'size' es negativo deja la lista intacta. Si 'size' es 0 entonces deja la lista vacía. Si size es mayor que el tamaño de la lista entonces deja la lista intacta. El método recorre la lista desde el final para dejar los 'size' primeros nodos en la lista.

```

public static <E> void trimToSize(PositionList<E> list, int size) {

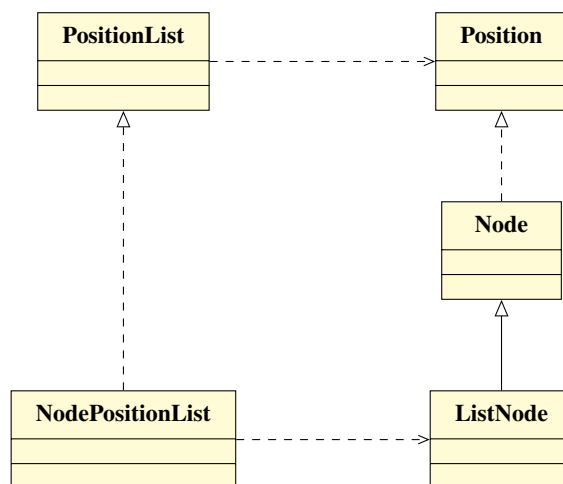
    if (list != null && !list.isEmpty() &&
        size >= 0 && size < list.size()) {

        Position<E> cursor = list.last();
        size = list.size() - size;
        while (size > 0) {
            Position<E> prv = list.prev(cursor);
            list.remove(cursor);
            cursor = prv;
            size--;
        }
    }
}

```

6.2 Implementación mediante cadenas de nodos doblemente enlazados

El diagrama de clases correspondiente a la implementación sería:



Clase Node<E> y ListNode<E>

La clase Node<E> implementa los objetos nodo para múltiples estructuras de datos. Implementa el interfaz Position<E> y por tanto provee código para el método 'element' declarado en dicho interfaz. Para su uso en listas se ha implementado la clase ListNode<E> que extiende de la clase Node<E>. El código fuente de Node<E>, ListNode<E> y de NodePositionList<E> se encuentra disponible en el Moodle de la asignatura.

Entre la clase Node<E, O> y ListNode<E> tienen los siguientes atributos y métodos:

- owner es un objeto que identificará unívocamente la lista a la que pertenece el nodo. Está declarado como 'final', lo que significa que una vez inicializado su valor por el constructor ya no podrá modificarse.
- prev y next son respectivamente referencias al nodo previo y al nodo siguiente en la lista.
- elem referencia el elemento que se almacena en el nodo.

Los métodos y constructores de Node<E>:

- La clase provee un constructor y también "getters" "setters" para 'prev', 'next' y 'elem'. (El método `element` es el "getter" de 'elem').
- Intencionalmente, no hay constructor de copia ni se sobrescribe `equals`.
- No hay "setter" para `owner` ya que este atributo no puede modificarse al ser declarado como **final**.
- No hay "getter" para `owner`. Solamente un objeto nodo puede determinar usando el método `kinOf` si otro objeto nodo tiene el mismo 'owner' ("kin" significa "pariente" en Inglés). Por ejemplo, dado un nodo 'n1' distinto de null y un nodo 'n2', la invocación `n1.kinOf(n2)` indicará si 'n1' y 'n2' tienen el mismo `owner`.
- Los "setters" `setPrev` y `setNext` de los atributos `prev` y `next` pueden tomar como parámetros válidos tanto el valor null como una referencia a un nodo con el mismo `owner`. En otro caso lanzan la excepción `IllegalArgumentException`. (¿Qué pasa si el nodo parámetro es 'this'?).

Como veremos en la clase `NodePositionList<E>`, permitir **null** tiene sentido si queremos borrar un nodo de la lista desenlazándolo de su nodo previo y de su nodo siguiente poniendo 'next' y 'prev' a null. Por otro lado el constructor también permite que los parámetros que se le pasan sean null. Uno de los constructores de la clase `NodePositionList<E>` tiene que crear un nodo con un atributo temporalmente a null para poder enlazar los nodos centinela.

Clase `NodePositionList<E>`

Tiene tres atributos:

- `size`, que almacena el tamaño o número de nodos de la lista, representado mediante un entero.
- Dos nodos centinela, `header` y `trailer`, que no almacenan elementos y son usados para marcar el principio y el fin de la lista.

El propósito de los nodos centinela es que al insertar/borrar siempre haya un nodo previo y otro siguiente al que enlazar/desenlazar. El código de inserción/borrado (métodos `addFirst`, `addBefore`, `remove`, etc.) resulta más sencillo y uniforme que si tuviéramos referencias al primer y último nodo útiles. Por ejemplo, cuando se inserta un nodo por primera vez, el nodo previo es el `header` y el nodo siguiente es el siguiente al 'header' que en esta ocasión es el nodo `trailer`. Si inmediatamente después se inserta otro nodo al principio de la lista, de nuevo el nodo previo es el 'header' y el nodo siguiente es el siguiente al 'header' que en esta ocasión es el nodo insertado anteriormente.

El tamaño de la lista *está acotado por la memoria* disponible, en concreto, por la memoria dinámica disponible para crear nuevos objetos nodo en los métodos de inserción 'addFirst', etc.

Idealmente el tipo del atributo 'size' debería ser concordante con la memoria dinámica disponible. Hemos considerado que 'int' es suficiente. En Java 'int' es un entero de 32 bits con valor máximo 2.147.483.647. Para los propósitos de esta asignatura es suficiente. También está disponible el tipo 'long' de enteros de 64 bits cuyo valor máximo es 9.223.372.036.854.775.807. La memoria dinámica disponible depende de la arquitectura y de otros factores circunstanciales como el número de procesos ejecutándose, etc. La correspondencia entre 'size' y la memoria dinámica (número de nodos que podemos insertar) no es una cuestión meramente teórica. Los enteros de tamaño fijo puede sufrir desbordamiento ("overflow"). Por ejemplo:

```
int i = 2147483647;
System.out.println(++i);
```

el valor mostrado por consola es -2147483648.

Por tanto, la inserción con éxito (hay memoria disponible) de un elemento en una lista con 2147483647 elementos conllevaría a un valor negativo de `size`. Esta problemática afecta a otras estructuras de datos. No volveremos a incidir en ella pero tenedla presente.

Algunos comentarios sobre los métodos de `NodePositionList<E>`:

- El primer constructor crea una lista vacía que consiste en 'header' y 'trailer' encadenados.
- El segundo constructor construye la lista tomando los elementos de un array. Usa el "for-each" de java que veremos en detalle en el tema de [Iteradores].
- El tercer constructor (constructor de copia) toma los elementos (¡no los nodos!) de otra lista. No se copian los elementos, se copian sus referencias, es decir, hace "shallow copy". Los constructores segundo y tercero no comprueban si el vector o la lista es null ya que en ese caso sí debe lanzarse `NullPointerException`.
- El primer constructor y los métodos de inserción 'addFirst', 'addLast', 'addBefore' y 'addAfter' son los únicos que crean nodos. Al crearlos los marcan como pertenecientes a la lista. En concreto, le pasan al constructor de nodos un entero generado por un método privado 'iayf' ("I am your father") que encapsula la forma de generar el entero con que marcar los nodos. (Si quisiéramos cambiar el método de marcado sólo tenemos que modificar el método 'iayf').
- En la implementación actual, el método `iayf` llama al método `hashCode` de la superclase de `NodePositionList`, esto es, de `Object`. Según la especificación del lenguaje Java, el método `hashCode` de `Object` genera un entero único para cada objeto de clase `Object`. Por tanto, el valor de `owner` de un nodo será un entero que identifica unívocamente el objeto lista al que pertenece. Observad que `NodePositionList<E>` podría

sobrescribir el método `hashCode` sin problema porque `iyaf` invoca explícitamente el `hashCode` de `Object` usando `super`. Como apunte sobre `hashCode` decir que el método `'hashCode'` de un objeto arbitrario debe generar un entero que puedan usar las tablas de dispersión ("hash tables") como valor de dispersión ("hash") del objeto.

<http://eclipsesource.com/blogs/2012/09/04/the-3-things-you-should-know-about-hashcode/>.

Incidiremos sobre esto más adelante en el tema de tablas de dispersión.

- El método privado `checkNode` comprueba si una posición es realmente un nodo válido de la lista, es decir:
 1. No es `'null'`.
 2. Es de clase `'Node<E>'`.
 3. Es un nodo de la lista (se comprueba haciendo `'kinOf'` con el `'header'`).
 4. Está encadenado a un nodo previo y a un nodo siguiente.
- El método `iterator`, los métodos sobrescritos `toString` e `equals`, así como el método `toArray` los explicaremos cuando veamos el tema de Iteradores. Simplemente observamos que el método `equals` compara `this` con cualquier objeto `'o'` que implemente el interfaz `PositionList<E>`, el cual no tiene que ser de clase `'NodePositionList'` necesariamente. El concepto de iterador nos permitirá implementar este tipo de comparación de igualdad general en otros TADs donde los métodos ofrecidos por el interfaz no son suficientes para recorrerlo. En el caso de las listas se podría haber implementado el `equals` general usando el cursor como en `'toString'` porque el interfaz `PositionList<E>` ofrece métodos para recorrer la lista.

Complejidad de los métodos de la clase `NodePositionList<E>`

Nos interesa la complejidad (coste) en tiempo (de ejecución) y espacio (consumo de memoria) en el "caso peor" (el peor escenario posible). A modo de recordatorio:

- $O(1)$: la complejidad en tiempo del método en el caso peor se mantiene constante independientemente de lo que aumente el tamaño de la lista
- $O(n)$: la complejidad en tiempo del método en el caso peor es linealmente proporcional al tamaño `'n'` de la lista. Es decir, si la lista tiene `'n'` elementos entonces la complejidad es `'n'`. Si el tamaño es `'n+1'` entonces la complejidad es `'n+1'`, etc.

La complejidad temporal en el caso peor de todos los métodos (excepto los de iteración) es $O(1)$. Por ejemplo, en la inserción y el borrado se le proporciona a los métodos `'addAfter'`, `'addBefore'` y `'remove'` el nodo con el elemento a insertar. Los métodos únicamente tienen que modificar los enlaces de los nodos, cuyo coste es constante respecto de (o "no depende de") el tamaño de la lista. Los métodos `'addFirst'` y `'addLast'` toman elementos, no nodos, pero crear un objeto nodo también tiene complejidad constante.

Ejercicio 6.2 Dado un elemento, si no se tiene su posición entonces habría que buscarla. Añadir a la clase e implementar el método:

```
public Position<E> getPos(E e)
```

que debe devolver la primera posición en la lista que contiene el elemento `'e'`, o `null` si el elemento no está en la lista. ¿Cuál sería la complejidad en el caso peor de dicho método? ■

Listas de la Java Collections Framework

La Java Collection Framework (JCF) es una librería de TADs estándar de Java

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>

Los interfaces importantes de la JCF son `Collection<E>` y `Map<K, V>`. A resaltar:

- Usado en muchas aplicaciones reales.
- El código fuente de las clases no está disponible y por tanto no podemos usarlo para docencia.
- Se ofrecen muchos métodos útiles: conversión a array y viceversa, reordenación (`'sort'`, `'shuffle'`, `'reverse'`, etc), computación con segmentos, búsqueda por patrones, etc.
- `Collection<E>` extiende `Iterable<E>` pero `'Map<K, V>'` no. Veremos las consecuencias de ésto en el tema de Iteradores.

Tradicionalmente los TADs (incluidos las listas) se usan para insertar, buscar y borrar elementos directamente o a través de índices, no se usan abstracciones como `'Position<E>'`. Sin embargo, `'Position<E>'` es un TAD útil que permite obtener una buena complejidad y además nos permite estudiar la implementación mediante cadenas de nodos

doblemente enlazados. La Java Collection Framework ofrece varios TADs lista. Ninguno usa 'Position<E>'. Se usan índices y/o elementos. Los interfaces de lista extienden `Collection<E>` que a su vez extiende `Iterable<E>`.

- Interfaces: `List<E>` (usa índices) y `ListIterator<E>`. Hay además clases abstractas intermedias: `AbstractList<E>`, `AbstractSequentialList<E>`, etc.
- Implementaciones: `ArrayList<E>`, `LinkedList<E>`, y otras más como pueden ser `AttributeList`, `CopyOnWriteArrayList`, `RoleList`, `RoleUnresolvedList`, `Vector`, etc.

Las dos más utilizadas serían: (1) `ArrayList<E>`, que implementa listas usando vectores cuyo tamaño puede “modificarse” (realmente se crea un vector nuevo del tamaño deseado y se copian elementos); (2) la clase `LinkedList<E>`, que es parecida a `NodePositionList<E>`, pero se usan directamente elementos (no `Position<E>`) y se permite el uso de índices, puesto que hereda de `List<E>`.

7. Iteradores

Muchos programas tienen que iterar sobre los elementos de un TAD). Por ejemplo, con las listas de posiciones, los programas iteran linealmente sobre la lista: posicionándose en el nodo 'first' (o el 'last'), avanzando dentro de un bucle con 'next' (o retrocediendo con 'prev'), y usando 'null' como valor centinela del bucle, junto con otras posibles condiciones. Los elementos se obtienen invocando el método 'element' sobre el objeto nodo correspondiente. Recordad lo visto en los recorridos de listas de posiciones. Por lo general una iteración sobre un TAD se realiza mediante bucles. Se invocan los métodos del interfaz del TAD para obtener elementos y moverse. (Esto es posible si hay suficientes métodos observadores que permitan realizar la iteración.)

Concepto de iterador

Ese código tiene varios problemas: (1) Es específico para listas de posiciones porque se usan datos y métodos específicos de su interfaz: cursor de tipo 'Position<E>', métodos 'first' y 'next', y uso de 'null' como centinela. No se puede adaptar el código fácilmente para iterar sobre otros TADs como, por ejemplo, colas FIFO; (2) el programador tiene que declarar el cursor, inicializarlo adecuadamente, e incrementarlo adecuadamente. Hay varias oportunidades para cometer errores y generar una 'IllegalArgumentException'.

Se puede abstraer el cursor y tener código fácilmente reusable para otros TADs convirtiendo el cursor en un TAD llamado Iterador.* Un objeto iterador permite iterar *linealmente* (uno a uno) los elementos de *otro* objeto TAD que almacena elementos, de forma que la iteración no se realiza usando los métodos del interfaz del TAD sino usando los métodos del interfaz del iterador y se puede reutilizar el código para iterar otros TADs.

Mostramos un ejemplo con 'show':

```
public static <E> void show(PositionList<E> list) {  
  
    Position<E> cursor = list.first(); // Inicializacion.  
  
    while (cursor != null) {                // Bucle mientras el cursor  
                                           // referencie a donde hay  
                                           // un elemento.  
  
        System.out.println(cursor.element()); // Se hace algo con el elemento.  
  
        cursor = list.next(cursor);          // Se avanza el cursor.  
    }  
}
```

He aquí una forma de abstraer el cursor en un objeto iterador:

```
public static <E> void show(PositionList<E> list) {  
  
    Iterator<E> it = list.iterator(); // La lista nos da un objeto iterador  
                                     // (un cursor) ya inicializado.  
  
    while (it.hasNext()) {                // Bucle mientras el cursor
```

```

// referencie a donde hay un elemento.

System.out.println(it.next()); // Se hace algo con el elemento
// y queda el cursor avanzado.
}
}

```

El método `it.next()` devuelve el elemento accesible desde el cursor (no devuelve un nodo), y además dejará el cursor avanzado, lo que constituye un *efecto secundario* o *side effect* parecido a un post-incremento del índice en la iteración de un vector:

```

int i = 0; // Inicialización

while (i < arr.length) { // Bucle mientras el índice
// está en rango

    System.out.println(arr[i++]); // Se hace algo con el elemento
// y queda el índice avanzado.
}

```

En la práctica, el objeto iterador puede mantener un cursor como atributo, pero habrá casos en los que no será necesario. El cursor podrá referenciar el elemento directamente, o un objeto que contiene un elemento, p.ej., el nodo de una lista, etc. Por ello he escrito “el cursor referencia a donde hay un elemento”. Ahora el código con el objeto iterador se puede adaptar fácilmente a otros TADs que implementen el interfaz `Iterable<E>`. Lo único que cambia es que se le pide el iterador a otro TAD en vez de a una lista, el resto permanece igual.

```

public static <E> void show(Iterable<E> tad) {

    Iterator<E> it = tad.iterator(); // El TAD nos da un objeto iterador
// (un cursor) ya inicializado.

    while (it.hasNext()) { // Bucle mientras el cursor
// referencie a donde hay un elemento.

        System.out.println(it.next()); // Se hace algo con el elemento
// y queda el cursor avanzado.
    }
}

```

Otra forma de escribir un bucle con iteradores usando un **for**:

```

for (Iterator<E> it = list.iterator(); it.hasNext(); ) {
    System.out.println(it.next());
}

```

No hay incremento en el bucle porque el método ‘next’ realiza el incremento como efecto secundario.

No deben usarse métodos modificadores (inserción o borrado) del interfaz del TAD durante la iteración (por ejemplo, añadir elementos al mismo tiempo que iteramos) pues podría dejarse el iterador en estado inconsistente.

Interfaces `Iterator<T>` e `Iterable<T>`

El interfaz `Iterator<E>` declara los métodos que debe implementar todo objeto iterador. El interfaz está en `java.util.Iterator`.

```

public interface Iterator<E> {
    public E next() ; // * obligatorio implementarlo */
    public boolean hasNext() ; // * obligatorio implementarlo */
    public void remove() ; // * no obligatorio y problemático! */
}

```

El interfaz ‘`Iterable<E>`’ declara un método ‘`iterator`’ que debe devolver un objeto iterador. Este interfaz es la pieza que nos permitirá asociar iteradores a un TAD. El interfaz está en ‘`java.lang.Iterable`’.

```

public interface Iterable<E> {
    public Iterator<E> iterator() ;
}

```

Cuidado con el significado de los métodos del iterador:

- Los nombres de los métodos `hasNext` y `next` suelen despistar.
- NO debe interpretarse `hasNext` como "¿hay siguiente al cursor?".
- NO debe interpretarse `next` como "dame el siguiente elemento al cursor".
- El método `hasNext` indica si el cursor del iterador referencia un elemento (o si referencia un objeto que contiene un elemento, p.ej., un nodo de una lista).

Una interpretación válida es: "¿referencia el cursor algo donde hay un elemento, el cual todavía no te he pedido con 'next'?". Usando la metáfora de una frutería, cuando el frutero dice "¿hay siguiente?" si hay alguien entonces levanta la mano el que está primero, cuando atiende al primero (`next`) y vuelve a decir "¿hay siguiente?", levanta la mano el que estaba segundo, etc. En una secuencia de elementos, el primer "siguiente" es el primer elemento. Otra posible interpretación válida de `hasNext` podría ser "¿es el cursor distinto de null?".

El método `next` guarda el elemento al que se llega a través del cursor, avanza el cursor (o lo deja a `null` si no se puede avanzar) y devuelve el elemento guardado. Una interpretación válida sería "dame el elemento del cursor y deja el cursor avanzado (o a `null` si no se puede avanzar más)". También hay un método `remove`, que borra el elemento que devolvió el `next` que se ha ejecutado inmediatamente antes. Los métodos `hasNext` y `next` se usan en bucles, donde `hasNext` es la condición centinela que debe comprobarse antes de invocar `next`. El método `remove` puede invocarse si antes (en el tiempo) se ha invocado un `next`.

Desde el cursor no puede alcanzarse ningún elemento del TAD cuando éste es vacío, ni cuando `next` devuelve el último elemento de la iteración. En este último caso `next` deja el cursor a `null` pues no hay elemento siguiente.

- **Ejemplo 7.1** Tenemos un TAD con 2 elementos 'A' y 'B'. Al crearse el objeto iterador, el cursor referencia el primer elemento porque el TAD no es vacío.

```
{A,B}
  ^
  |
cursor

hasNext() devuelve true.
next()   avanza el cursor a B y devuelve A.

{A,B}
  ^
  |
cursor

hasNext() devuelve true.
next()   avanza el cursor a null y devuelve B.

{A,B}

cursor -> null

hasNext() devuelve false.
next()   lanza NoSuchElementException.
```

Al crearse el objeto iterador:

- Si el TAD está vacío:

```
hasNext() devuelve false.
next()   lanza NoSuchElementException.
```

- Si el TAD no está vacío:

```
hasNext() devuelve true.
next()   avanza el cursor al segundo elemento (si hay) o a null (si no
hay) y devuelve el primer elemento.
```

Se pueden usar varios iteradores simultáneamente sobre un mismo TAD:

```

for (Iterator<E> it1 = list.iterator(); it1.hasNext(); ) {
    E e = it1.next();
    for (Iterator<E> it2 = list.iterator(); it2.hasNext(); it2.next()){
        System.out.println(e);
    }
}

```

Método remove del iterador

El método `remove` del iterador debe borrar del TAD el elemento que ha sido devuelto por el último `next`. El método `remove` debe lanzar `IllegalStateException` cuando se invoca pero no se invocó ningún `next` antes en el tiempo (no necesariamente en la línea anterior).

```

if (!it.hasNext())
    it.remove();    // Incorrecto, no hay elementos.

if (it.hasNext())
    it.remove();    // Incorrecto, no hay 'next' previo.

while (it.hasNext()) {
    it.next();
    it.remove();    // Correcto
}

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove();        // Correcto si no falló el 'next' anterior

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove()         // Correcto si no falló el 'next' anterior
it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove();        // Correcto si no falló el 'next' anterior

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
it.remove();        // Correcto si no falló el 'next' anterior

it.next();          // Correcto si 'hasNext' es cierto, sino incorrecto.
.                  ^
.                  // Mucho código aquí sin ningún 'remove' |
.                  |
it.remove();        // Correcto si no falló el 'next' anterior ----/

it.remove();        // Incorrecto, no hay 'next' previo.

```

Si usamos un iterador sobre un TAD, sólo está permitido borrar con el `remove` del iterador.

Ejemplos de uso de iteradores

■ Ejemplo 7.2 Método que devuelve el décimo elemento de una lista (contando desde 1)

```

public static <E> E tenth(PositionList<E> list)
throws NoSuchElementException {
    if (list.size() < 10) throw new NoSuchElementException();
    else {
        Iterator<E> it = list.iterator();
        for (int i = 1; i < 10; i++) {
            it.next();
        }
        return it.next();
    }
}

```


■ Ejemplo 7.3 Método que borra todos los elementos de una lista

```
public static <E> void deleteAll(PositionList<E> list) {
    Iterator<E> it = list.iterator();
    while (it.hasNext()) {
        it.next();
        it.remove();
    }
}
```

Recordad que el método `remove` debe invocarse después de `next` y que `remove` borra el nodo donde está el elemento que ha devuelto el `next` anterior. ■

■ Ejemplo 7.4 Método que devuelve la suma de los elementos de una lista de enteros. Vamos a ver dos versiones:

(1) Versión que asume que los elementos nunca son null:

```
public static int sumaElems(PositionList<Integer> list) {
    Iterator<E> it = list.iterator();
    int suma = 0;
    while (it.hasNext()) {
        suma += it.next();
    }
    return suma;
}
```

(2) Versión que asume que los elementos pueden ser null. Se necesita una variable donde guardar el elemento:

```
public static int sumaElems(PositionList<Integer> list) {
    Iterator<E> it = list.iterator();
    int suma = 0;
    while (it.hasNext()) {
        Integer e = it.next();
        if (e != null) suma += e;
    }
    return suma;
}
```

■ Ejemplo 7.5 Método que devuelve el máximo elemento de una lista de enteros asumiendo que los no son null.

```
public static int maximum(PositionList<Integer> list)
throws EmptyListException {
    if (!it.hasNext()) throw new EmptyListException();
    int m = it.next();
    while (it.hasNext()) {
        m = Math.max(m, it.next());
    }
    return m;
}
```

■ Ejemplo 7.6 Método que indica si un elemento no null está en una lista

(1) Versión sin iterador:

```
public static <E> boolean member(E e, PositionList<E> list) {
    Position<E> cursor = list.first();
    while (cursor != null && !e.equals(cursor.element())) {
        cursor = list.next(cursor);
    }
}
```

```
// cursor == null || cursor != null && e.equals(cursor.element())
return cursor != null;
}
```

(2) La siguiente implementación con iterador es **errónea**:

```
public static <E> boolean member(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    while (it.hasNext() && !e.equals(it.next()))
        ; // el "incremento" del cursor lo hace 'next'

    // !it.hasNext() || it.hasNext() && e.equals(it.next())
    return it.hasNext();
}
```

Para saber por qué es errónea ejecutad el código en papel sobre una lista con un único nodo con elemento igual a 'e'. Recordad que 'it.next()' produce un efecto secundario: también avanza el cursor.

(3) La siguiente implementación correcta utiliza una variable booleana 'found' para guardar el estado de la última comparación.

```
public static <E> boolean member(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    boolean found = false;
    while ( it.hasNext() && !(found = e.equals(it.next())) )
        ;

    // !it.hasNext() || it.hasNext() && found
    return found;
}
```

■ **Ejemplo 7.7** Método que elimina de una lista la primera aparición del un elemento. Para simplificar asumimos que el elemento no es null.

```
public static <E> void borra(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    boolean found = false;
    while ( it.hasNext() && !(found = e.equals(it.next())) )
        ;
    if (found) it.remove();
}
```

■ **Ejemplo 7.8** Método que elimina todas las apariciones de ese elemento no null

```
public static <E> void borra(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    while (it.hasNext()) {
        if (e.equals(it.next())) {
            it.remove();
        }
    }
}
```

Ahora una solución que contempla el caso de que los elementos sean null

```
public static <E> void borra(E e, PositionList<E> list) {
    Iterator<E> it = list.iterator();
    while (it.hasNext()) {
        E elem = it.next();
        if (e == null && elem == null || e != null && e.equals(elem)) {
```

```

        it.remove();
    }
}

```

■ **Ejemplo 7.9** Método que indica si una lista es sublista de otra

```

public static <E> boolean sublist(PositionList<E> l1, PositionList<E> l2) {
    if (l1 == null || l2 == null) return false;
    if (l1 == l2) return true;
    Iterator<E> it1 = l1.iterator();
    boolean res = false;
    while ( it1.hasNext() && (res = this.member(it1.next(),l2)) )
        ;
    return res;
}

```

■ **Ejemplo 7.10** Método que indica si dos listas son iguales. Recordad que dos listas son iguales si contienen los mismos elementos de izquierda a derecha.

(1) Versión con varias variables auxiliares:

```

public static <E> boolean iguales (PositionList<E> list1,
                                   PositionList<E> list2) {

    if (list1 == list2) return true;
    // El caso anterior incluye list1 == null && list2 == null
    if (list1 == null || list2 == null || list1.size() != list2.size())
        return false;

    Iterator<E> it1 = list1.iterator();
    Iterator<E> it2 = list2.iterator();
    E e1, e2;
    boolean iguales = true;

    while (it1.hasNext() && iguales) {
        e1 = it1.next();
        e2 = it2.next();
        iguales = e1 == null && e2 == null || e1 != null && e1.equals(e2);
    }

    return iguales;
}

```

(2) Creamos un método auxiliar para comparar elementos:

```

private static <E> boolean igualesElem(E e1, E e2) {
    return e1 == null && e2 == null || e1 != null && e1.equals(e2);
}

public static <E> boolean iguales (PositionList<E> list1,
                                   PositionList<E> list2) {

    .
    .
    .
    Iterator<E> it1 = list1.iterator();
    Iterator<E> it2 = list2.iterator();
    E e1, e2;
    boolean iguales = true;

```

```

    while (it1.hasNext() && iguales) {
        e1      = it1.next();
        e2      = it2.next();
        iguales = igualesElem(e1,e2);
    }

    return iguales;
}

```

(3) Eliminamos las variables elemento y subimos la comparación al bucle:

```

public static <E> boolean iguales (PositionList<E> list1,
                                  PositionList<E> list2) {
    .
    .
    .
    Iterator<E> it1 = list1.iterator();
    Iterator<E> it2 = list2.iterator();
    boolean iguales = true;

    while (it1.hasNext() && (iguales = igualesElem(it1.next(),it2.next())))
        ;
    return iguales;
}

```

No podemos eliminar la variable booleana, pues ésta guarda el estado de la última comparación. Necesitamos dicho estado porque `next` avanza el cursor. El siguiente código NO funciona:

```

while (it1.hasNext() && igualesElem(it1.next(),it2.next()))
    ;
// !it1.hasNext() || it1.hasNext && !igualesElem(it1.next(),it2.next())
return !it1.hasNext();

```

Este tampoco:

```

while (it1.hasNext() && igualesElem(it1.next(),it2.next()))
    ;
// !it1.hasNext() || it1.hasNext && !igualesElem(it1.next(),it2.next())
return igualesElem(it1.next(),it2.next());

```

Probadlo con dos listas que contienen un único elemento distinto el uno del otro.

¿Cuándo y cómo usar iteradores?

Los iteradores se usan para iterar sobre TADs que son colecciones de elementos. No todos los TADs serán colecciones de elementos y por tanto no todos serán iterables por iteradores. También hay TADs que son colecciones de elementos pero dada su naturaleza no son iterables (p.ej., conjuntos no acotados que no tienen “getters” para obtener elementos). Y si son iterables es con restricciones, como que el iterador puede ser no-determinista: recorre los elementos de forma arbitraria según cómo estén disponibles en el objeto que implementa del TAD. Pero entonces la computación que se realice con los elementos iterados debe dar el mismo resultado para cualquier posible permutación de todos los elementos. Por ejemplo, la suma de elementos de un conjunto de enteros daría el mismo resultado para cualquier permutación. Sin embargo mostrar elementos por pantalla daría resultados diferentes.

Para usar iteradores el problema debe requerir únicamente el acceso a elementos (método `next`) y/o borrar el último elemento devuelto por el iterador (método `remove`). No se pueden usar iteradores si hay que borrar un elemento que no sea el último devuelto por `next` (que es el elemento que borra el `remove` del iterador), o si es necesario acceder, por ejemplo, a los nodos que se recorren, puesto que `next` sólo devuelve el elemento y no tenemos acceso al nodo.

Ejercicio 7.1 Para ver cuando “no” usar iteradores, intentad programar con un iterador el `addBeforeElement` visto en el Ejemplo 6.2, es decir, un método que inserta un elemento “a” justo antes que el elemento “e”:

```

<E> void addBeforeElement (PositionList<E> list, E e, E a) {

```

El interfaz `Iterator<E>` sólo permite avanzar y borrar. Hay más interfaces en la JCF que extienden este interfaz con métodos para retroceder, reset, etc. En la JCF los interfaces de lista extienden `Collection<E>` que a su vez extiende `Iterable<E>`. También hay mecanismo “fail-fast” para múltiple iteradores cuando se modifica la colección durante la iteración [Goodrich].

Iteración con bucle “for-each”

El bucle *for-each* es una abstracción de código que permite recorrer cualquier estructura de datos iterable de forma homogénea.

Vamos a reescribir ‘show’ una vez más, pero en vez de:

```
public static <E> void show(PositionList<E> list) {
    Iterator<E> it = list.iterator();
    while (it.hasNext())
        System.out.println(it.next());
}
```

ahora usamos un *for-each*:

```
public static <E> void show(PositionList<E> list) {
    for (E e : list)
        System.out.println(e);
}
```

El *for-each* tiene una notación parecida a la cuantificación universal (\forall) de la lógica: “para todo elemento ‘e’ de tipo ‘E’ en el *iterable* ‘list’ ejecuta ‘System.out.println(e)’”.

La sintaxis del *for-each* es la siguiente:

```
for (type elem : expr) { stmts }
```

Donde:

- La variable `elem` tiene tipo ‘type’ y no aparece en la expresión ‘expr’.
- La expresión `expr` tiene tipo `Iterable<T>` o tipo “array de T”, con T un subtipo de `type`. Es decir, se puede usar *for-each* con arrays o con objetos iterables (objetos de clases que implementan el interfaz `Iterable<E>`).
- Dentro de `stmt` no se tiene acceso al iterador (a una variable que referencie el objeto iterador), sólo al elemento `elem`. No se pueden invocar `next`, `hasNext`, ni `remove`. Los dos primeros son invocados por el bucle de forma automática.

El *for-each* se recorre el TAD iterable **por completo** “for-each = para todo elemento”, etc. No debe usarse *for-each* para recorridos que pueden ser parciales. Por ejemplo, no debe usarse *for-each* para indicar si un elemento está en un TAD pues el recorrido debe terminar al encontrarse el elemento.

El compilador traduce el *for-each* a un “for.”rdinario con iteradores:

```
for (Iterator<E> it = list.iterator(); it.hasNext(); ) {
    E e = it.next();
    System.out.println(e);
}
```

La variable `it` local al bucle la genera el propio compilador.

■ Ejemplo 7.11 Suma de elementos de una lista de enteros, los elementos pueden ser null:

```
public static int sumaElems(PositionList<Integer> list) {
    int suma = 0;
    for (Integer e : list)
        if (e != null)
            suma += e;
    return suma;
}
```

Por ejemplo, dos de los constructores de la clase `NodePositionList` están escritos con *for-each*.

■ Ejemplo 7.12 Reescribimos el `toString` de la clase `NodePositionList` usando *for-each*:

```
public String toString() {
    String s = "[";
```

```

    for (E e : this) {
        if (e == null)
            s += "null";
        else
            s += e.toString();
        if (cursor != last())
            s += ", ";
    }
    s += "];";
    return s;
}

```

De manera informal podríamos verlo de la siguiente forma:

1. Cuando *expr* es de tipo *Iterable<T>* el bucle se lee como “ejecuta ‘stmt’ para todo elemento *elem* del iterable *expr*”. En este caso el bucle *for-each* es una abreviatura de:

```

    for (Iterator<type> it = expr.iterator(); it.hasNext(); ) {
        type elem = it.next();
        stmts
    }

```

donde ‘it’ es una variable oculta nueva que no aparece en ‘stmts’. Nótese que ‘elem’ se declara después de usarse ‘expr’ y por eso ‘elem’ no puede ocurrir en ‘expr’. Si no hay elementos para iterar (‘hasNext’ devuelve falso) no se ejecuta el código del *for-each*. La variable ‘elem’ puede declararse localmente dentro del bloque del bucle ‘for’. También se puede expresar el *for-each* con un bucle ‘while’:

```

    {
        Iterator<type> it = expr.iterator();
        type elem;
        while (it.hasNext()) {
            elem = it.next();
            stmts
        }
    }

```

java

2. Cuando *expr* es de tipo *array de T* entonces el bucle *for-each* es una abreviatura de:

```

    for (int j = 0; j < expr.length; j++) {
        type elem = v[j];
        stmts
    }

```

Donde *j* es una nueva variable que no aparece en ‘stmts’.

■ Ejemplo 7.13 Sumar los elementos de un vector:

```

public int sumaArray(int [] v) {
    int suma = 0;
    for (int e : v)
        suma += e;
    return suma;
}

```

■ Ejemplo 7.14 Ejemplo de MAL uso: mostrar los ‘n’ primeros elementos de un array de caracteres ‘v’. Usando *for-each* se recorre innecesariamente todo el array:

```

for (char c : v) {
    if (n > 0) {
        System.out.println(c);
        n--;
    }
}

```

Usar 'break' es un error de estilo y va contra la razón de ser de *for-each* que sirve para recorrer todo el vector:

```

for (char c : v) {
    if (n == 0) break;           // Suspense
    if (n > 0) {
        System.out.println(c);
        n--;
    }
}

```

La solución idónea es no usar el *for-each*. ■

Cómo implementar iteradores

Hay dos alternativas para implementar iteradores sobre TADs:

1. El objeto iterador itera ("mueve el cursor") **usando los métodos del interfaz del TAD** para obtener el siguiente elemento. De esta forma el iterador puede usarse para iterar sobre objetos de cualquier clase que implemente el interfaz. En otras palabras, dado el interfaz 'I', el iterador puede iterar sobre cualquier clase que implemente 'I' si usa únicamente métodos de 'I' para "mover" el cursor.
2. El objeto iterador itera ("mueve el cursor") **accediendo a los atributos de la clase que implementa el TAD**. Estos iteradores únicamente pueden usarse para iterar sobre objetos de las clases concretas. En otras palabras, dada la clase 'C' que implementa el interfaz 'I', el iterador definido para objetos de 'C' sólo puede usarse sobre objetos de 'C' y no sobre objetos de otras clases que implementan 'I'.

La primera alternativa es más deseable desde el punto de vista de abstracción y reusabilidad, pero puede no ser posible realizarla porque no hay suficientes métodos observadores en el interfaz, o porque la iteración usando métodos del interfaz es ineficiente.

Iteradores sobre interfaces A continuación damos una *receta* para implementar iteradores sobre TADs de forma que los iteradores invocan los métodos de los interfaces de los TADs.

1. El interfaz del TAD debe extender 'Iterable<E>' para indicar que las clases que implementen el interfaz tendrán que implementar el método 'iterator', el cual deberá devolver un objeto iterador (un objeto que implementa el interfaz 'Iterator<E>') para dicho TAD.

```

import java.util.Iterator;
public interface TAD<E> extends Iterable<E> {

    ... /* métodos del TAD */

    public Iterator<E> iterator() ;
}

```

Recordad que el interfaz `PositionList<E>` extiende `Iterable<E>`.

2. Se implementa una clase iterador cuyos métodos usan los métodos del interfaz del TAD para "mover el cursor":

```

import java.util.Iterator;
public class TADIterator<E> implements Iterator<E> {
    TAD<E> tad; /* el TAD sobre el que itera es un atributo */
    CursorTAD<E> cursor;

    public TADIterator(TAD<E> t) {
        tad = t;
        ... /* código aquí que inicializa el valor del cursor */
    }
    public boolean hasNext() { /* código aquí */ }
    public E next() { /* código aquí */ }
}

```

```

    public void remove()          { /* código aquí */ }
}

```

El constructor del objeto iterador toma como parámetro el objeto TAD sobre el que debe iterar. El tipo de 'cursor' depende del TAD concreto. Puede ser directamente un elemento:

```
E cursor;
```

También puede ser una referencia a un objeto que contiene un elemento (como en el caso de `PositionListIterator.java`, donde tenemos:

```
Position<E> cursor;
```

También puede ser un valor de algún tipo, a partir del cual se puede localizar el elemento en el TAD, etc.

3. Implementar el método 'iterator' en todas las clases que implementan el interfaz 'TAD<E>'. Dicho método devuelve un objeto iterador.

```

import java.util.Iterator;
public class TADClass1<E> implements TAD<E> {
    ...
    public Iterator<E> iterator() { return new TADIterator<E>(this); }
}

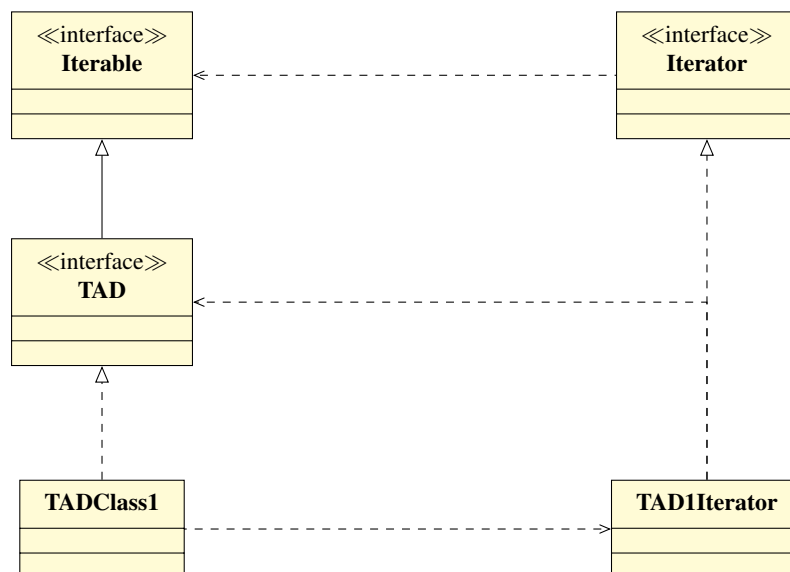
```

```

import java.util.Iterator;
public class TADClass2<E> implements TAD<E> {
    ...
    public Iterator<E> iterator() { return new TADIterator<E>(this); }
}

```

El diagrama de clases sería:



Los interfaces y clases del diagrama deben ser declarados dentro de un paquete.

El fichero `PositionListIterator.java` contiene la implementación del iterador para el interfaz `PositionList<E>`:

- Esta clase implementa `Iterator<E>` para `PositionList<E>`.
- El cursor no referencia un elemento de tipo 'E' sino una posición `Position<E>` porque los métodos del interfaz `PositionList<E>` tales como `next` y `prev` (que se usan para "mover el cursor") toman como parámetros posiciones, no elementos.
- `NodePositionList<E>` implementa el método `iterator` y usa el iterador en el método `equals`.

Iteradores sobre clases: A continuación se describe otra posibilidad para implementar iteradores utilizando para ello la clase sobre la que se quiere iterar, en lugar de utilizar el interfaz. **El material de esta sección es opcional y no será evaluado en el examen de teoría.** Incluimos el material para los alumnos que desean profundizar en la materia.

1. Se puede iterar sobre un TAD declarando el método 'iterator()' como método propio de TAD<E> sin extender de Iterable<E>:

```
import java.util.Iterator;
public interface TAD<E> {
    ...
    public Iterator<E> iterator();
}
```

Las clases que implementen el interfaz no tendrán que implementar 'iterator'. Sin embargo esto no es recomendable porque TAD<E> no es Iterable<E>. No podemos usar TAD<E> en contextos donde se espera un iterable, por ejemplo, no podemos hacer un *for-each* con el interfaz TAD<E>, aunque sí podríamos hacerlo con la implementación concreta del interfaz.

2. Se puede asociar un iterador a una clase en vez de a un interfaz:

- Código del interfaz 'TAD<E>':

```
public interface TAD<E> /* no extiende Iterable<E> */ { ... }
```

- Código de una clase:

```
public class TADClass1<E> implements TAD<E>, Iterable<E> {
    ...
    public Iterator<E> iterator() {
        return new TADClass1Iterator<E>(this);
    }
}
```

- Código del iterador asociado a la clase TADClass1<E>:

```
public class TADClass1Iterator<E> implements Iterator<E> {
    TADClass1<E> tadClass1;
    CursorTAD<E> cursor;

    public TADClass1Iterator(TADClass1<E> t) {
        tadClass1 = t;
        ... /* código aquí que inicializa el valor del cursor */
    }

    public boolean hasNext() { /* código aquí */ }
    public E next() { /* código aquí */ }
    public void remove() { /* código aquí */ }
}
```

- El constructor del iterador toma como parámetro un objeto de la clase TADClass1<E>, no un objeto de tipo TAD<E>. Los métodos hasNext y next pueden implementarse o bien usando los métodos del interfaz TAD<E> (porque TADClass1<E> implementa los métodos de TAD<E>) o bien accediendo directamente a los atributos de TADClass1<E> si son públicos.

3. Finalmente, una variación de lo anterior es definir TADClass1Iterator<E> como una clase anidada no estática ("non-static") dentro de TADClass1<E>, pudiendo la primera acceder a los atributos privados de la segunda. Consultar en la documentación de Java los conceptos de clases *nested*, *inner*, *local*, y *anonymous*, así como la siguiente URL:

<http://download.oracle.com/javase/tutorial/java/javaOO/nested.html>

IMPORTANTE: En los dos últimos casos (2 y 3) el iterador TADClass1Iterator<E> sólo puede usarse con objetos de clase TADClass1<E>. No puede usarse con objetos de otras clases que implementen TAD<E>. Esas clases deben tener su propio iterador.

Iteradores con alguna propiedad

Los TAD's vistos hasta ahora permiten recorrer los elementos contenidos en una estructura de datos siguiendo el orden "oficial" de la estructura, es decir, en una lista el iterador que se obtiene llamando al método iterator() del TAD recorre todos los elementos de la lista manteniendo el orden en el que se encuentran en la lista.

Pero también nos pueden interesar diferentes vistas del mismo TAD, por ejemplo, me puede interesar recorrer los elementos de la lista en sentido inverso o alterno, o me puede interesar recorrer únicamente los elementos que cumplan una cierta propiedad, p.e. lo que no sean null, o los positivos, ...

Esto se puede hacer creando nuevas clases que implementen el interfaz `Iterator<E>` proporcionándole al nuevo iterador las propiedades buscadas. Por ejemplo el siguiente iterador recorrerá la `PositionList list` en orden inverso:

```
public class PositionListReverseIterator<E> implements Iterator<E> {

    private PositionList<E> list;
    private Position<E> cursor;

    public PositionListReverseIterator(PositionList<E> list) {
        if (list == null) {
            throw new IllegalArgumentException ("La lista es null");
        }
        this.list = list;
        this.cursor = list.last();
    }

    public boolean hasNext() {
        return cursor != null;
    }

    public E next() throws NoSuchElementException {
        if (cursor == null) throw new NoSuchElementException();

        E e = cursor.element();
        cursor = list.prev(cursor);
        return e;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

La idea fundamental detrás de esto, sería, dada la misma estructura de datos, dependiendo del iterado utilizado, disponer de diferentes vistas o formas de recorrer la estructura. Hay que tener en cuenta que estos iteradores no se crean mediante el método `iterator()` en el TAD, sino que deben ser creados a mano teniendo una referencia a la estructura de datos:

```
PositionList<String> l = new NodePositionList<String>(...);
Iterator<String> it = new PositionListReverseIterator<String>(l);

// Devolverá los elementos en orden inverso
while(it.hasNext()) {
    System.out.println(it.next());
}
```

8. Recursividad

Un método es recursivo cuando se invoca a sí mismo en su propia definición. Un ejemplo típico y sencillo es el método que calcula el 'factorial' de un entero positivo. El factorial de un entero positivo $n > 0$ se define como el valor de la expresión $n * (n - 1) * \dots * 1$. Para $n \leq 0$ el factorial se define como 1. Mostramos dos posibles soluciones iterativas. Ambas usan una variable acumulador, en la que se acumulan las multiplicaciones. La primera solución itera desde 'n' hasta '2'. La segunda desde '2' hasta 'n'.

```
public static int factorial(int n) {
    int r = 1;
    while (n > 1) {
        r = n * r;
        n--;
    }
    return r;
}
```

```
public static int factorial(int n) {
    int r = 1;
    for (int i = 2; i <= n; i++) {
        r = i * r;
    }
    return r;
}
```

Dejemos a un lado el problema de que en Java el tipo 'int' tiene un rango de valores limitado (probad qué pasa por ejemplo con `factorial(100)`). Dejemos también a un lado cómo solucionarlo (¿qué otros tipos primitivos o clases se podrían usar?).

Una definición recursiva es más intuitiva y elegante.

Observad que la expresión $(n - 1) * (n - 2) * \dots * 1$ calcula `factorial(n-1)`. Por tanto se puede definir `factorial(n)` en función de `factorial(n-1)`:

$$\underbrace{n * \underbrace{(n-1) * (n-2) * \dots * 1}_{\text{factorial}(n-1)}}_{\text{factorial}(n)}$$

$$\text{factorial}(n) = n * \text{factorial}(n-1)$$

En código:

```
public static int factorial(int n) {
    if (n <= 1) return 1;
    else return n * factorial(n-1);
}
```

```
}
```

que queda más escueto usando una expresión condicional:

```
public static int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n-1);
}
```

Cualquier programa iterativo tiene uno equivalente recursivo y viceversa. Observad que en el ejemplo del factorial, la condición del 'if' en la versión recursiva es la negada de la condición del 'while' en la versión iterativa, y que llamarse recursivamente es una forma de iterar.

Este código tendría la siguiente ejecución para `factorial(4)`:

```
factorial(4)
4 * factorial(3)
4 * 3 * factorial(2)
4 * 3 * 2 * factorial(1)
4 * 3 * 2 * 1
4 * 3 * 2
4 * 6
24
```

En el diagrama anterior, observamos que las llamadas recursivas se van *apilando*. Una vez se ha llegado al caso base, se calcula la multiplicación al retornar de la llamada recursiva, así hasta llegar a la primera invocación.

La recursión se suele implementar utilizando la pila de la memoria (donde se almacenan los registros de activación de los métodos) pero hay implementaciones más eficientes de la recursión como la transformación a estilo en paso de continuaciones (*continuation-passing style* o CPS) que deja los métodos recursivos con recursión de cola (*tail recursion*) que a su vez puede optimizarse en un bucle.

Mostramos cómo podría optimizar un compilador el método 'factorial'. Primero recordamos la versión original:

```
public static int factorial(int n) {
    if (n <= 1) return 1;
    else return n * factorial(n-1);
}
```

Se invoca `factorial` recursivamente sobre '`n-1`'. Cuando éste retorna el valor, se multiplica por '`n`'. Sin embargo, se podría invocar `factorial` recursivamente pero pasándole además un parámetro extra que indique qué tiene que hacer cuando termine de computar el factorial de `n-1`. Es decir, se podría invocar 'factorial' diciéndole: (1) Computa el valor del factorial de `n-1`; (2) después multiplica ese valor por `n` y entonces retorna el resultado. De esta forma `factorial` tendría recursión de cola: la llamada recursiva sería el último mandato del método y no habría que multiplicar después.

Para que la recursión tenga sentido, la definición debe tener uno o más **casos base** (no recursivos) y debe haber un orden bien fundado en los valores de los argumentos, de forma que las (una o más) llamadas recursivas se realizan sobre valores anteriores en el orden, llegándose eventualmente a alguno de los casos base. En caso contrario el método recursivo no termina (es equivalente a un bucle infinito). En el ejemplo del factorial el orden bien fundado es el de los enteros positivos ($n > n-1 > \dots > 1$) y se llega al caso base.

Por ejmplo, sas siguientes variantes son incorrectas:

```
public static int factorial(int n) {
    return n * factorial(n-1);           // no hay caso base.
}

public static int factorial(int n) {
    if (n == 1) return 1;
    else return n * factorial(n-2);     // no alcanza el caso base
}
```

El primer ejemplo es equivalente a un bucle infinito. Si el compilador implementa la recursión con una pila, puesto que el computador no tiene memoria infinita, eventualmente se producirá un error de desbordamiento de pila o *stack overflow*. Con la optimización de recursión de cola no habrá desbordamiento y se quedará colgado como en un bucle infinito.

Mostramos el código equivalente iterativo de los ejemplos anteriores:

```

public static int factorial(int n) {
    int r = 1;
    while (n != 1) {
        r = n * r;
        n -= 2;
    }
    return r;
}

public static int factorial(int n) {
    int r = 1;
    while (true) {
        r = n * r;
        n--;
    }
    return r;
}

```

■ **Ejemplo 8.1** El cálculo del máximo común divisor (también su versión iterativa):

```

public static int mcd(int n, int m) {
    if (m == 0) return n;
    else return mcd(m, n % m);
}

public static int mcd(int n, int m) {
    while (m != 0) {
        int tmp = m;
        m = n % m;
        n = tmp;
    }
    return n;
}

```

■ **Ejemplo 8.2** Ejemplo con más de un caso base y llamada recursiva (aquí hemos obviado los 'else'):

```

public static int fib(int n) {
    if (n < 0) return 0;
    if (n <= 1) return n; /* dos casos base */
    return fib(n-1) + fib(n-2);
}

```

■ **Ejemplo 8.3** Búsqueda binaria de un elemento en un vector ordenado. Para realizar la recursión necesitamos un método auxiliar que lleva cuenta del rango en el que se está buscando. La búsqueda se llama “binaria” porque en cada paso hay sólo dos (binario) opciones: o bien se busca en el rango desde `start` hasta el elemento `m-1` anterior al central, o bien se busca en el rango desde el elemento `m+1` siguiente al central hasta `end`. Al comienzo, `start = 0` y `end = arr.length-1`.

Versión iterativa:

```

public static boolean member(int elem, int [] arr) {
    if (arr == null || arr.length == 0 ||
        elem < arr[0] || elem > arr[arr.length - 1])
        return false;
    int start = 0;
    int end = arr.length - 1;
    int m;

```

```

while (start <= end && arr[(m = (start + end) / 2)] != elem) {
    if (elem < arr[m])
        end = m-1;
    else
        start = m+1;
}
// start > end || start <= end && arr[m] == elem
return start <= end;
}

```

Ahora la versión recursiva:

```

public static boolean member(int elem, int [] arr) {
    if (arr == null || arr.length == 0 ||
        elem < arr[0] || elem > arr[arr.length - 1]) /* no es caso base */
        return false;
    else
        return memberRec(elem, arr, 0, arr.length-1);
}

private static boolean memberRec(int elem, int [] arr,
                                   int start, int end) {

    if (start > end) return false; /* caso base */
    int m = (start + end) / 2;
    if (elem == arr[m]) return true; /* caso base */
    if (elem < arr[m]) return memberRec(elem, arr, start, m-1);
    return memberRec(elem, arr, m+1, end);
}

```

El método principal no es recursivo, únicamente trata los casos frontera e invoca el método auxiliar (y por ello privado) recursivo que toma todos los parámetros necesarios para resolver el problema. Podemos simplificar el número de casos base en uno:

```

private static boolean memberRec(int elem, int [] arr,
                                   int start, int end) {

    if (start >= end) return elem == arr[start];
    else {
        int m = (start + end) / 2;
        if (elem <= arr[m])
            return memberRec(elem, arr, start, m);
        else
            return memberRec(elem, arr, m+1, end);
    }
}

```

Observad que al unificar los dos casos base, en la primera llamada recursiva *m* no puede decrementarse, de lo contrario el código sería incorrecto.

Recursión sobre listas de posiciones

Las listas se pueden definir de forma recursiva: una lista de posiciones puede definirse de forma recursiva como una secuencia de nodos que es o bien vacía (caso base) o bien está formada por un nodo seguido de una secuencia de nodos. Los métodos recursivos sobre listas explotarán esta definición.

■ Ejemplo 8.4 El método `show` que muestra por pantalla los elementos de una lista

Versión iterativa:

```

public static <E> void show(PositionList<E> list) {
    if (list != null) {
        Position<E> cursor = list.first();
        while (cursor != null) {

```

```

        System.out.println(cursor.element());
        cursor = list.next(cursor);
    }
}

```

Versión recursiva. El método principal no es recursivo, únicamente trata los casos frontera e invoca el método auxiliar (y por ello privado) recursivo. Los casos base se dan cuando el parámetro cursor es 'null' (caso en que la lista es vacía y caso en que se ha llegado al final de la lista).

```

public static <E> void show(PositionList<E> list) {
    if (list != null) showRec(list, list.first());
}

private static <E> void showRec(PositionList<E> list, Position<E> cursor) {
    if (cursor == null) return;
    else {
        System.out.println(cursor.element());
        showRec(list, list.next(cursor));
    }
}

```

Como el método recursivo no devuelve ningún valor, es habitual no hacer uso del 'return' explícito y dejar únicamente la rama que hace algo distinto de simplemente retornar.

```

private static <E> void showRec(PositionList<E> list, Position<E> cursor) {
    if (cursor != null) {
        System.out.println(cursor.element());
        showRec(list, list.next(cursor));
    }
}

```

Se puede subir el caso base de "lista vacía" del método recursivo auxiliar al método principal para ahorrarse una llamada recursiva:

```

public static <E> void show(PositionList<E> list) {
    if (list != null && !list.isEmpty())
        showRec(list, list.first());
}

```

■ Ejemplo 8.5 El método toString de la clase NodePositionList<E>

Versión iterativa:

```

public String toString() {
    String s = "[";
    for (Position<E> cursor = first(); cursor != null; cursor = next(cursor)) {
        if (cursor.element() == null)
            s += "null";
        else
            s += cursor.element().toString();
        if (cursor != last()) s += ", ";
    }
    s += "]";
    return s;
}

```

Versión recursiva con método auxiliar privado:

```

public String toString() {
    return "[" + toStringRec(first()) + "]";
}

private String toStringRec(Position<E> cursor) {

```

```

    if (cursor == null) return "";
    String s;
    if (cursor.element() == null)
        s = "null";
    else
        s = cursor.element().toString();
    if (cursor != last())
        s += ", ";
    return s + toStringRec(next(cursor));
}

```

Versión anterior con expresiones condicionales:

```

public String toString() {
    return "[" + toStringRec(this.first()) + "]";
}

private String toStringRec(Position<E> cursor) {
    if (cursor == null) return "";
    return (cursor.element() == null ? "null" : cursor.element().toString())
        + (cursor != last() ? ", " : "")
        + toStringRec(this.next(cursor));
}

```

Versión recursiva con parámetro de acumulación:

```

public String toString() {
    return toStringRec("[" , first());
}

private String toStringRec(String acum, Position<E> cursor) {
    if (cursor == null) return acum + "]";
    if (cursor.element() == null)
        acum += "null";
    else
        acum += cursor.element().toString();
    if (cursor != last())
        acum += ", ";
    return toStringRec(acum, list.next(cursor));
}

```

El código anterior puede simplificarse separando los casos de lista vacía y final de lista:

```

public String toString() {
    if (isEmpty()) return "[]";
    else return toStringRec("[" , first());
}

private String toStringRec(String acum, Position<E> cursor) {
    if (cursor.element() == null)
        acum += "null";
    else
        acum += cursor.element().toString();

    if (cursor == last())
        return acum + "]";
    else
        return toStringRec(acum + ", ", list.next(cursor));
}

```

■ **Ejemplo 8.6** Método que suma todos los elementos de una lista de enteros considerando que los elementos

pueden ser null:

```
public static int sumaElems(PositionList<Integer> list) {
    if (list == null) return 0;
    else return sumaRec(list, list.first());
}

private static int sumaRec( PositionList<Integer> list
                           , Position<Integer> cursor) {
    if (cursor == null)
        return 0;
    else if (cursor.element() == null)
        return sumaRec(list, list.next(cursor));
    else
        return cursor.element() + sumaRec(list, list.next(cursor));
}
```

Evitamos una llamada recursiva subiendo el caso base al método principal:

```
public static int sumaElems(PositionList<Integer> list) {
    if (list == null || list.isEmpty()) return 0;
    else return sumaRec(list, list.first());
}
```

También podemos simplificar el método recursivo:

```
private static int sumaRec( PositionList<Integer> list
                           , Position<Integer> cursor) {
    if (cursor == null) return 0;
    return (cursor.element() == null ? 0 : cursor.element())
        + sumaRec(list, list.next(cursor));
}
```

Otra versión que utiliza una variable auxiliar donde se acumula la suma (veremos el parecido con la versión con parámetro de acumulación de la siguiente sección):

```
private static int sumaRec( PositionList<Integer> list,
                           Position<Integer> cursor) {
    int r = 0;
    if (cursor != null)
        r = (cursor.element() == null ? 0 : cursor.element())
            + sumaRec(list, list.next(cursor));
    return r;
}
```

■ Ejemplo 8.7 Método que suma con parámetro de acumulación (recursión de cola):

```
public static int sumaElems(PositionList<Integer> list) {
    if (list == null || list.isEmpty()) return 0;
    else return sumaRec(0, list, list.first());
}

private static int sumaRec( int r
                           , PositionList<Integer> list
                           , Position<Integer> cursor) {
    if (cursor == null)
        return r;
    else if (cursor.element() == null)
        return sumaRec(r, list, list.next(cursor));
    else
        return sumaRec(r + cursor.element(), list, list.next(cursor));
}
```

El código del método recursivo auxiliar puede simplificarse:

```
private static int sumaRec( int r
                          , PositionList<Integer> list
                          , Position<Integer> cursor) {
    if (cursor != null)
        r = sumaRec( r + (cursor.element() == null ? 0 : cursor.element())
                    , list
                    , list.next(cursor))
    return r;
}
```

■ Ejemplo 8.8 Borrar una aparición de un elemento distinto de null

```
public static void borra(PositionList<E> list, E elem) {
    if (list != null) borraRec(list, elem, list.first());
}

private static void borraRec( PositionList<E> list,
                             , E elem
                             , Position<E> cursor) {
    if (cursor != null) {
        if (cursor.element() != null && cursor.element().equals(elem))
            list.remove(cursor);
        else
            borraRec(list, elem, list.next(cursor));
    }
}
```

■ Ejemplo 8.9 Borrar todas las apariciones de un elemento distinto de null

```
public static void borra(PositionList<E> list, E elem) {
    if (list != null) borraRec(list, elem, list.first());
}

private static void borraRec( PositionList<E> list
                             , E elem
                             , Position<E> cursor) {
    if (cursor != null) {
        Position<E> aux = cursor;
        cursor = list.next(cursor);
        if (aux.element() != null && aux.element().equals(elem))
            list.remove(aux);
        borraRec(list, elem, cursor);
    }
}
```

Código alternativo del método recursivo:

```
private static void borraRec( PositionList<E> list
                             , E elem
                             , Position<E> cursor) {
    if (cursor != null) {
        Position<E> nxt = list.next(cursor);
        if (cursor.element() != null && cursor.element().equals(elem))
            list.remove(cursor);
        borraRec(list, elem, nxt);
    }
}
```

```
}
```

- **Ejemplo 8.10** Inserción ordenada en una lista ordenada de enteros. Para simplificar asumimos que los elementos no son null:

```
public static void insertOrden(PositionList<Integer> list, E elem) {
    if (list != null) insertRec(list, elem, list.first());
}

private static void insertRec( PositionList<Integer> list
                               , E elem
                               , Position<E> cursor) {

    if (cursor == null)
        list.addLast(elem);
    else if (elem <= cursor.element())
        list.addBefore(cursor, elem);
    else
        insertRec(list, elem, list.next(cursor));
}
```